AN ARRAY PROCESSOR WITH A LARGE NUMBER OF PROCESSING ELEMENTS

By

Nelson Castro Machado

January 1, 1972

CAC Document No. 25

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

CAC Document No. 25

UIUCDCS-R-72-499

AN ARRAY PROCESSOR WITH A LARGE NUMBER OF PROCESSING ELEMENTS

By

Nelson Castro Machado

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

January 1, 1972

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign and supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the U.S. Army Research Office-Durham under Contract No. DAHC04-72-C-0001.

ABSTRACT

This paper describes a new type of array processor (SPEAC) which could be characterized as an intermediate between ILLIAC IV and the Associative Processor. The number of processing elements (PE's) is typically 1K but could go as high as 8K. Each PE is a relatively simple unit with about 1K equivalent gates, designed to allow implementation either on a single very complex LSI chip or on several MSI chips. Each PE plus its memory (PEM) could then be assembled on one single printed circuit board or ceramic substrate.

Processing is performed in groups of four bits which allows variable word length. Maximum freedom in data format and instruction format is made possible by the use of a microprogrammable control unit (CU). Therefore, the machine is quite versatile and can be used efficiently either on floating-point large precision problems (matrix operations, signal processing, etc.) or on fixed-point small precision ones (character manipulation, picture processing, etc.).

PE design is carried out in great detail and a general sketch of the CU is presented. Operations are described and timed, with particular emphasis on floating-point addition (20 μsec per PE for 32 bits) and floating-point multiplication (25 μsec per PE for 32 bits). A few typical applications are presented along with their time estimates.

ACKNOWLEDGMENTS

I wish to express my deep gratitude to Professor Daniel L. Slotnick for the constant aid and encouragement in every phase of the research herein described.

My colleague Robert L. Mercer is to be thanked for the many helpful discussions and suggestions.

I would also like to express my appreciation for the financial support given by the ILLIAC IV Project and the Center for Advanced Computation of the University of Illinois.

Finally, special thanks are due to Suzanne Sluizer for the efficient and accurate typing of the final document, to Fred Hancock and Jose Martinez for their careful execution of many complex drawings, and to my wife Arlene for unmatched patience and constant incentive.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Faster computers may be obtained either by improving the raw speed of the circuits and components or by adopting a better organization, i.e., using the same circuits in a more efficient architecture. Indefinite improvements in circuit speed cannot be expected due to fundamental physical constants, the most obvious of these being the speed of light. Therefore, new approaches to computer organization must be found if projected demands of computer users are to be met, particularly in the area of large scientific problems.

In recent years, a fair amount of attention has been given to non-conventional organizations and the first two super-computers utilizing these new concepts will become operational within a few months: the pipeline processor CDC-STAR [1] and the array computer ILLIAC IV [2] [3]. Several other approaches have been proposed in the literature, deserving special mention the parallel processor, extensively studied by IBM [4], and the associative processor, a type of array processor utilizing an associative memory and distributed logic [5]. Goodyear Aerospace Corporation has been working on an associative processor and successful tests have been performed on a reduced scale prototype.

An endless number of questions, discussions and comparisons can and have been raised when the capabilities and handicaps of the different organizations are considered [6]. As usual, one can usually find a specific application in which a given architecture excels and a pathological case in which the same approach fails miserably. It is not the purpose of this paper to engage in such comparisons. It will instead deal only with a particular

organization: the array computer.

The array processor family of computers has been widely accepted by the computer community as a cost-effective approach in a particular but rather important set of applications. In the sequel, this type of architecture is examined and a new approach to the design of an array processor is proposed in order to take advantage of recent and contemplated developments in the fields of LSI circuits and solid state memories.

2. THE ARRAY COMPUTER AND ITS APPLICATIONS

2.1 General Description of an Array Computer

ILLIAC IV will be taken here as the "typical" array computer. This section is not supposed to be a complete description of ILLIAC IV and a certain familiarity with [2] and [3] is assumed. Only a few basic concepts are considered here in order to set the stage for the following discussion.

Figure 1 shows the functional diagram of a classical computer. It consists of: 1) A memory to hold operands and instructions, 2) A control unit that fetches instructions from the memory, decodes them and issues control signals to 3) An arithmetic unit that performs the operations on operands taken from the memory. The most radical approach to parallelism would obviously be to duplicate the elements shown in Figure 1 a number (n) of times providing adequate interconnections between the elements. This is the multiprocessor or parallel processor approach. Although powerful, this organization leads to several implementation problems and seems to be impractical for large n. (The Burroughs B6500 uses this organization with $n_{max} = 4$.) One of these problems is the economic burden caused by the multiplicity of control units since in a sophisticated classical machine the control unit accounts for rather more than fifty percent of the total gate count. This leads to the array computer approach, whose functional diagram is shown in Figure 2. Only arithmetic units and memories are duplicated and one single control unit (CU) drives the "array" of arithmetic units. Actually not the whole control unit can be made central since certain control decisions are operand-dependent (normalization for example). Therefore, a minimum amount of control is kept local and each arithmetic unit plus its local control will

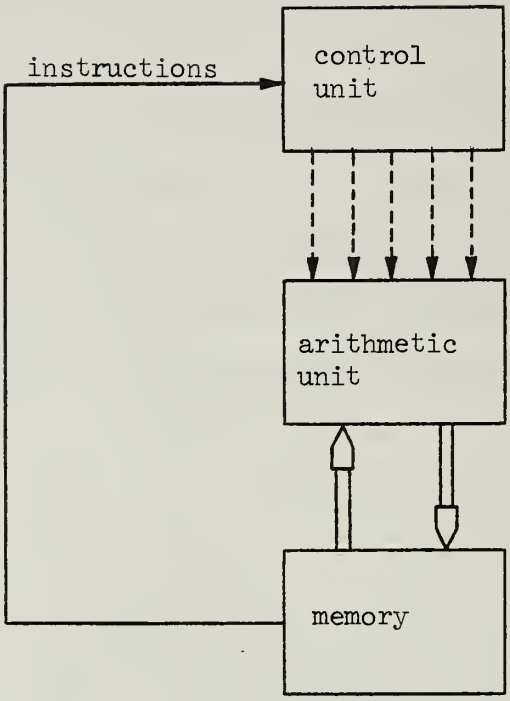


Figure 1. A Classical Computer

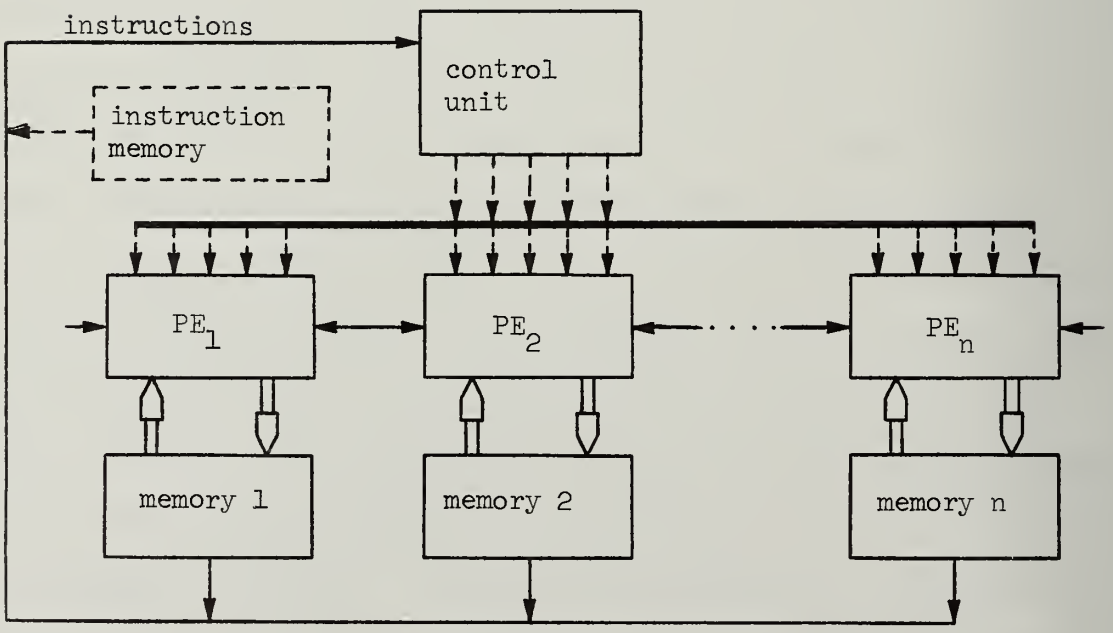


Figure 2. An Array Computer

be called processing element (PE). Each PE operates on its own memory (PEM). The term processing unit (PU) will be used to designate a PE with its PEM. Instructions can be stored either across the PEM's or in a special instruction memory.

Therefore, an array computer is characterized by the fact that a single instruction stream is executed simultaneously by at the most n PU's. The concepts of local indexing, routing and mode control will now be introduced.

The biggest restriction imposed by this type of organization is obviously that every PE must be performing precisely the same instruction on the same addresses on its own PEM. These constraints can be relaxed to a good extent with the introduction of extra hardware to allow: a) local indexing: each central base address, "broadcast" by the CU to each PE, is locally indexed, b) mode control: each instruction is locally modified by the PE's. The simplest form of mode control is to locally decide if central instruction "I" will be locally executed as "I" or as a no-op; i.e., each PE can be turned on or off. This is the only type of mode control available in ILLIAC IV (extreme mode control capability would obviously lead to a multiprocessor approach), c) routing: obviously, for most applications, at a certain point in the computation PE_i may need an operand which is stored in PEM_j, $i \neq j$. Therefore, some way of "routing" operands from one PE to another is highly desirable. The most complete freedom of routing would be obtained if a cross-bar switch were provided linking each PEM to each PE. Naturally, this solution is prohibitively expensive for large values of n. The simplest type of routing is to link PE_i to PE's i-1 and i+1. This is called "neighbor routing." Obviously, non-neighbor routing is obtained with

a sequence of neighbor routings.

2.2 Typical Applications and Their Requirements

The obvious application for an array computer is on problems in which the same operations must be repeated over a set of operands. Matrix operations fit nicely in this category and therefore this type of machine will work well on solving systems of linear equations, Fourier transforms, systems of partial differential equations, etc. Several areas of major scientific interest are included in such formalizations and the best known proposed applications for an array computer are: weather analysis and prediction, linear programming, seismic data processsing, hydrodynamic flow analysis, phased array radar processing, picture processing, etc.

Since a new type of array processor was contemplated, the first step was to elaborate a list of questions about the features of an array processor and submit it to several users in different areas of applications. In this way an opinion could be formed as to which features are needed for each application and which compromises would be acceptable.

Users in four areas of application were interviewed: 1) weather problem (WP), 2) seismic signal processing (SP), 3) linear programming (LP), and 4) hydrodynamic flow problem (HP).

The basic questions asked were:

a) How much floating-point operations does your application need? Could you do with fixed-point only?

b) What precision is needed for your application? How many bits is the typical precision in the input data?

c) How important is local indexing in your application? To

which extent is local indexing used only as a solution to poor routing facilities?

d) How much routing is done? Would only neighbor routing be sufficient? What are typical numbers for non-neighbor routing?

e) Mention any other problems encountered and facilities desired in your area of application.

It should be pointed out that all persons interviewed are ILLIAC IV users. ILLIAC IV contains 64 extremely powerful PE's with a complete repertoire of floating and fixed point instructions. Words are 64 bits long and can be used in submultiple precision variants of two 32-bit words or eight 8-bit words. There are facilities for local indexing and routing (accomplished through an optimal combination of distance 1 (neighbor) routings and distance 8 routings). Mode control is on-off only.

The following facts were established by the survey above:

a) Floating point: Floating point seems to be a luxury turned necessity. All users admitted that they could probably do without floating point by careful scaling of the quantities. They also admitted that they would hate to be forced to do that. The consensus is that presently a viable machine should have, if not hardware floating-point instructions, at least a good, fast set of floating-point subroutines.

b) Precision: Naturally, the precision requirements are heavily dependent on the particular application and method of solution: WP uses 32-bit words although the initial data has a typical precision of 8 bits only. It is felt that performing computation on 32-bit words is good insurance against precision erosion

due to severe numerical error propagation with the methods presently used. SP receives data from sensors in 13 to 14 bits precision and operates on 32-bit mode. Incidentally, simple format conversion of the input data accounts for a considerable amount of processing time in this application. SP could conceivably be performed with less precision than 32 bits: 18 or 24 bits should be adequate. LP is the application with the heaviest requirements on precision: I/O is performed in 32-bit mode but internal calculations use 64 bits to avoid severe error buildup in LP problems with about 400 equations. In fact, even 64-bit precision is inadequate for larger problems and the use of multiple precision routines is envisioned. HP has been using 32 bits which is adequate for low precision inputs. However, 48 to 64 bits would be ideal for future applications. Finally a few special but important applications need much less precision. Picture processing can be done with 4 to 8 bits of precision and a recently developed area--linear programming with Boolean variables--uses 1-bit precision for the variables and "small" integers for the coefficients.

The conclusion is obvious: a versatile machine should have as many precision modes as possible. This was the case with serial by bit machines which featured variable word length. Speed requirements forced the introduction of parallel processing of a word and the variable word convenience and efficiency was lost except for some low-precision instruction variants as the ones featured in ILLIAC IV.

c) Local Indexing: This seems to be a very important feature, heavily used by almost all application. Its main use is definitely to avoid slow routings in a "skewed" type of matrix storage [3]. However, a few other types of use for local indexing did appear.

d) Routing: Routing is the most difficult problem in an array computer. Complete and unlimited routing facilities are economically impossible for large values of n. The ILLIAC IV approach did satisfy its users, however. Definitely the most frequent type of routing is neighbor routing. Odd routing distances do appear, however, in a few important cases: table look-ups and log-sums (i.e., the problem of obtaining $\sum_{i=1}^{n} a_i$ where each a_i is stored in a different PE) are two examples.

2.3 Considerations on the Number and Complexity of the PE's

The array-processor family of computers has at present two well established members: ILLIAC IV and the Associative Processor (AP). Both these machines were extensively studied and are actually being built. In a sense, however, they represent two extremes in this design philosophy: ILLIAC IV has a relative small (64) number of PE's, each an extremely powerful floating-point word-parallel unit with 13K gates. The AP, described in [5], has a very large number (on the order of 2^{12} - 2^{15}) of PE's, each an extremely simple fixed-point serial-by-bit unit containing only 32 gates. Each ILLIAC IV PE has a floating point add time of 175 nsec. and a floating-point multiply time of 225 nsec. for 32-bit operands. The AP has a fixed-point add time of 35 μsec. and a fixed point multiply time of approximately 1 msec. for 32-bit

operands. Therefore, a 12K PE AP could add fixed point about as fast as ILLIAC IV. Multiplication would still be much slower (about 20 times slower even for a 12K PE AP). Routing capability in the AP is extremely limited: only neighbor routing is permitted, on a bit-by-bit basis. PEM is 2K 64-bit words long in ILLIAC IV and only 256 bits long in the AP. However the AP's PEM is an associative memory allowing simultaneous interrogation of n bits (n is the number of PE's). Obviously, ILLIAC IV's conventional PEM's could also be considered as an associative memory allowing simultaneous interrogation of 64 words.

It seems obvious that the AO is a much less versatile machine than ILLIAC IV, i.e., its field of application is quite limited. However, it may come as a surprise that in the problems to which it is well suited (especially radar tracking applications), the AP is quite cost-effective. In fact, its proponents argue that it can perform those special jobs at the same rate as ILLIAC IV but at 1/30th of the cost.

A few generalizations are in order: One could consider a set of array computer M_1, M_2, ... , M_n each with a simpler (slower) PE than its predecessor but with a larger number of PE's in order to keep constant the average speed. Figure 3 illustrates the number of PE's x speed of each PE for these machines. Figures 4 and 5 represent some rough qualitative estimates about the versatility of these machines (i.e., how large is the set of applications for which they are well suited, i.e., can compute approximately n times faster than a sequential machine with same speed as each PE) and the cost-efficiency of such machines for such suitable problems. The estimate in Figure 4 is practically obvious: the sequential machine (n=1) is the most versatile. As n grows, the number of problems that the machine can handle

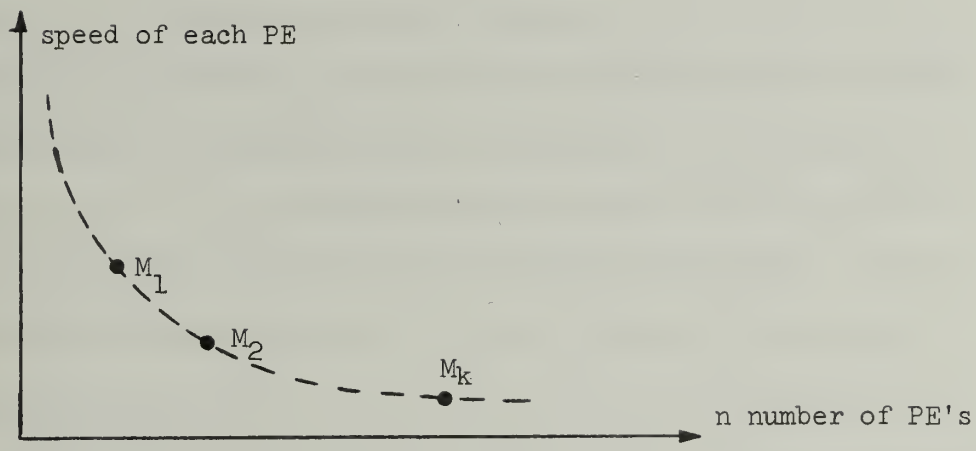


Figure 3. A Family of Array Computers with Constant Average Speed

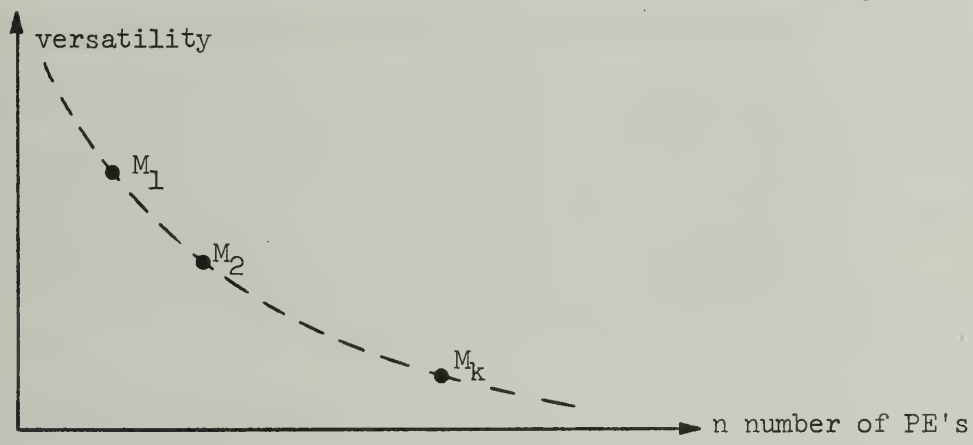


Figure 4. Versatility as a Function of the Number of PE's

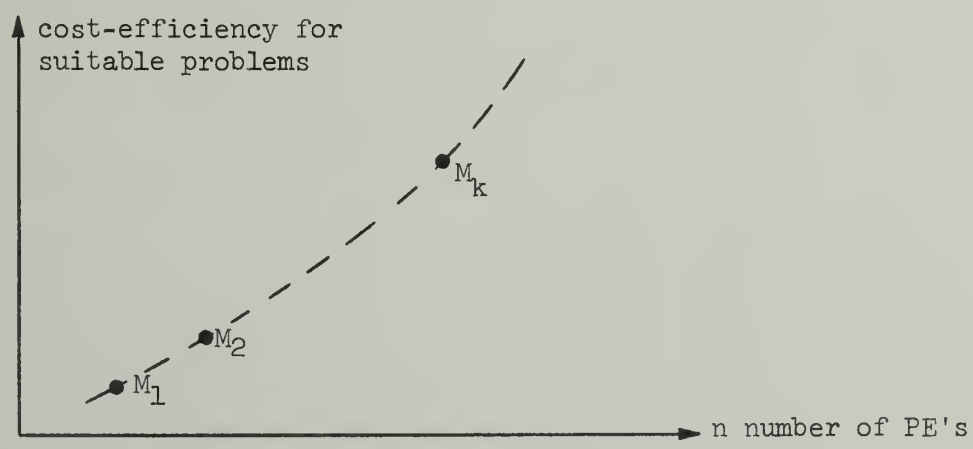


Figure 5. Cost efficiency as a Function of the Number of PE's

efficiently obviously decreases. Figure 5 is harder to justify. In fact, it is a guess based in two extremes: ILLIAC IV and the AP. A third machine, however, to be introduced later, does seem to verify this hypothesis: as n grows and each PE is simplified, modern integrated circuit techniques (LSI) allow a very rapid decrease in the cost per PE.

These considerations justify the idea of exploring the possibilities of a third type of array computer: the SPEAC (for small PE Array Computer). This machine would be between the AP and ILLIAC IV in number of PE's and PE power and hopefully would achieve a happy compromise between ILLIAC IV's relative versatility and the AP's cost-efficiency. The initial goals were:

$$n_{SPEAC} \sim 10\, n_{ILL\ IV} \quad \text{to } 100\, n_{ILL\ IV}$$

$$\text{PE speed}_{SPEAC} \sim \frac{1}{10} \text{ PE speed}_{ILL\ IV} \quad \text{to } \frac{1}{100} \text{ PE speed}_{ILL\ IV}$$

$$\text{gates per PE}_{SPEAC} \sim \frac{1}{10} \text{ gates per PE}_{ILL\ IV} \quad \text{to } \frac{1}{100} \text{ gates per PE}_{ILL\ IV}$$

The remainder of this paper is dedicated to exploring the feasibility and characteristics of this new machine.

3. SPEAC's HARDWARE

Initially, a few general considerations are made in order to establish the design goals that dictated the structure chosen for the hardware. The multiplication algorithm is also presented as a preface to the actual hardware description since the PE has been specifically designed to implement this algorithm efficiently.

3.1 General Considerations

a) The PE will be simple enough and built in a quantity high enough to warrant the expense of building special-purpose MSI to LSI integrated circuits. At first, it was hoped that a whole PE could be contained in a single LSI chip. This still seems to be possible, at least with the kind of technology foreseeable within a decade: a bipolar integrated chip with density on the order of 1 to 2K equivalent gates would be needed. However, even if one does not count on such extremes of built-to-order LSI, the proposed design could be implemented using a few dozen standard or nearly standard MSI chips, allowing an entire PU to be packed in one printed circuit card.

b) The results of the survey mentioned in Section 2.2 indicate the need of some floating-point capability. Naturally, entirely hardware-implemented floating-point is out of the question in a simple PE. However, the hardware should allow efficient implementation of floating-point routines. Serial processing, by bit or by groups of bits is the only way to keep the gate count low. This leads naturally to variable word length as a means

of satisfying the conflicting precision requirements outlined in Section 2.2.

c) Most contemplated applications have a high frequency of multiplications, typical of scientific problems. Therefore, multiplication should be as fast as possible, ideally almost as fast as addition as is the case in the ILLIAC IV PE.

d) Due to the existence of a CU, the PE must be strictly synchronous and local control must be minimized. Any synchronism or data-dependent optimization is wasted since the CU must always wait for the worst-case which almost certainly occurs for large n. This rules out certain classical methods like: increasing the speed of multiplication by adding only when the multiplier bit is one and simply shifting when it is zero. Instead, the CU must always output micro-orders for the worst-case and:

   either: the method is such that the extra operations are no-ops for non-worst-case conditions (example: add on a zero multiplier bit);

   or: some local control (typically a flip-flop) will inhibit certain steps in non-worst-case conditions (example: normalization, recomplementation).

e) An accumulator is impractical in a variable word length machine since it would have to be as long as the worst-case-length. Therefore, variable word length machines are typically 2- or 3-address machines. Three addresses are quite desirable since they avoid the frequent duplication of operands (to avoid its

destruction) found in 2-address machines. The classical shortcoming of 3-address machines, unnecessarily large instructions when the third address is equal to a previous one, can easily be avoided by adopting a variable length instruction format. Therefore, each instruction (op-code) will have a large number of variants with different lengths, from a minimum of zero addresses (in this case the old contents of the address registers would be used as addresses) to a maximum of six addresses, three basic addresses plus three addresses for local indexing. Word length of each operand and of the result might also be specified in the address part. The resulting instruction format is illustrated in Figure 6.

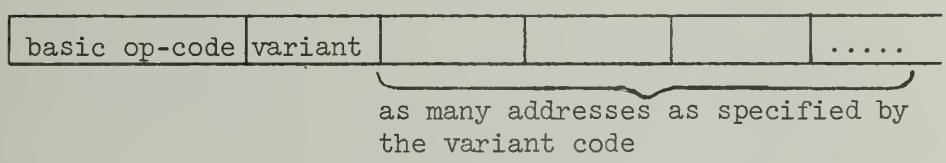


Figure 6. Instruction Format

f) Timing considerations: In order to satisfy the initial estimates set forth in Section 2.3, an addition time of 3 to 30 μsec and a multiplication time of 4 to 40 μsec are needed. Considering the basic PEM cycle time of the order of one-half μsec (this assumption will be explained in Section 3.2), and noticing that 1 to 3 PEM's cycle times (depending on the amount of interleave) are required per serial operation of the PE, one concludes that a 30 to 60 μsec addition time is obtained in a bit-by-bit PE for

32-bit fixed point addition. Straight multiplication will take 32 times as much or about 1 msec. This is far too slow and a serial by hexadecimal digit PE (i.e., serially processes chunks of 4 bits) is now considered. Addition time (32 bits, fixed point) now goes down to 8 to 15 μsec which is convenient. Straight multiplication, taking 32 times longer, is still quite slow. The next step would be a serial by byte PE but this presents two problems: firstly, normalization is either rather complicated and slow or it is done in 8-bit increments causing an unacceptable erosion in precision; secondly, the number of gates in the PE will be quite larger. Therefore, a serial by hexadecimal digit PE seems to be the best compromise: normalization in 4 bit increments (i.e., exponent base = 16) is quite acceptable and widely used in present computers. A somewhat elaborate multiplication algorithm (described in the next section) will be adopted to bring the multiplication time down to acceptable values.

g) Since the basic unit of data in the PE is one hexadecimal digit instead of a whole word, the machine is capable of accepting several different word formats provided the CU is able to generate an appropriate microsequence for that format. This immediately suggests the idea of micro-programming. Therefore, no particular word format will be picked and the PE control wire set will be chosen as carefully as possible in order to maximize the number of formats and operations that can be dealt with by writing adequate micro-programs at the CU level. The variable

format feature can be quite useful in certain applications (like seismic signal processing) in which format conversion accounts for a significant percentage of the processing time.

Summing up, the following design goals are thus established for SPEAC:

- PE built with MSI and LSI integrated circuits.
- One printed circuit card per PU.
- Variable word length.
- Multiplication not much slower than addition.
- Up to 3 addresses (possibly indexed) per instruction.
- Variable instruction length.
- PE serial by hexadecimal digits.
- Variable word format.
- Microprogramming capability.

3.2 The Multiplication Algorithm

As pointed out in Section 3.1, "straight" multiplication techniques (i.e., bit-by-bit) yield an unacceptably high multiplication time as compared to the addition time. On the other hand, ver-high-speed multiplication of the type used in ILLIAC IV requires a massive increase in the number of gates. The best compromise for SPEAC seems to be some form of hexadecimal multiplication algorithm allowing multiplication times roughly proportional to N^2 where N is the number of hexadecimal digits in the operands rather than the number of bits. It is also required that the algorithm be able to generate the product without the need to store double precision partial products since the PE has no register capable of holding long numbers and storing partial products in the memory will be slow and require the use of a portion of PEM as "scratchpad area."

The following multiplication algorithm satisfies the requirements above and is proposed for SPEAC: Consider the multiplication of two numbers A and B, each containing n+1 hexadecimal digits:

$$A = a_0 + a_1 2^4 + a_2 2^8 + \ldots + a_i 2^{4i} + \ldots + a_n 2^{4n} \tag{1}$$

$$B = b_0 + b_1 2^4 + b_2 2^8 + \ldots + b_i 2^{4i} + \ldots + b_n 2^{4n} \tag{2}$$

The double precision product M will be written as:

$$M = A \times B = m_0 + m_1 2^4 + m_2 2^8 + \ldots + m_i 2^{4i} + \ldots + m_{2n+1} 2^{4(2n+1)} \tag{3}$$

multiplying (1) and (2) as polynomials:

$$M = A \times B = a_0 b_0 + (a_0 b_1 + a_1 b_0) 2^4 + (a_0 b_2 + a_1 b_1 + a_2 b_0) 2^8 + \ldots + \left(\sum_{j=0}^{i} a_j b_{i-j} \right) 2^{4i} + \ldots + \left(\sum_{j=0}^{n} a_j b_{n-j} \right) 2^{4n} + \ldots + \left(\sum_{j=i}^{2n} a_{j-n} b_{n-j+i} \right) 2^{4i} + \ldots + a_n b_n 2^{4(2n)}$$

or:

$$M = \sum_{i=0}^{n} \left(\sum_{j=0}^{i} a_j b_{i-j} (2^{4i}) \right) + \sum_{i=n+1}^{2n} \left(\sum_{j=i}^{2n} a_{j-n} b_{n-j+i} (2^{4i}) \right) \tag{4}$$

From (3) = (4):

$$m_0 = (a_0 b_0)_{\text{mod } 16} \; ; \; c_0 = (a_0 b_0)_{\text{div } 16} \quad (\text{i.e., } a_0 b_0 = c_0 (2^4) + m_0)$$

$$m_1 = (c_0 + a_0 b_1 + a_1 b_0)_{\text{mod } 16} \; ; \; c_1 = (c_0 + a_0 b_1 + a_1 b_0)_{\text{div } 16}$$

$$\begin{cases} \vdots \\ m_i \quad = \left(c_{i-1} + \sum_{j=0}^{i} a_j b_{i-j} \right) \text{mod } 16 \; ; \; c_i = \left(c_{i-1} + \sum_{j=0}^{i} a_j b_{i-j} \right) \text{div } 16 \\ \; i \leq n \\ m_i \quad = \left(c_{i-1} + \sum_{j=i}^{2n} a_{j-n} b_{n-j+i} \right) \text{mod } 16 \; ; \; c_i = \left(c_{i-1} + \sum_{j=i}^{2n} a_{j-n} b_{n-j+i} \right) \text{div } 16 \\ \; i > n \\ \vdots \end{cases} \tag{5}$$

$$m_{2n} = (c_{2n-1} + a_n b_n)_{\text{mod } 16} \; ; \; c_{2n} = (c_{2n-1} + a_n b_n)_{\text{div } 16}$$

$$m_{2n+1} = c_{2n}$$

Therefore, the product may be computed as follows:

- multiply a_0 and b_0, the two low order digits of A and B; the result has two hexadecimal digits: $c_0(2^4) + m_0$; m_0 is the low order bit of the product and can be stored (in double precision multiplication) or discarded; c_0 is kept in an accumulator.
- multiply: $a_0 \times b_1$; add to the accumulator;
  multiply: $a_1 \times b_0$; add to the accumulator; the accumulator then contains $c_1 m_1$; store or discard m_1 and keep c_1 in the accumulator
  :
  and so on, using the equations (5) to determine each c_i and m_i.

It is easy to see that $(n+1)^2$ pairs of hexadecimal digits must be multiplied to compute the product of two numbers each with (n+1) hexadecimal digits. It should also be noticed that if a single precision product is desired, the product can replace one of the operands: m_0, m_1, ... , m_{m-1} are computed only to accumulate the carry and discarded. m_n is the first digit that may be in the final product and can be stored either "on top" of a_0 or b_0 since these two digits are not needed anymore to form the product. Finally, m_{2n} replaces a_n (or b_n). If $m_{2n+1} = c_n = 0$, then the product is stored correctly. However, if $m_{2n+1} = c_n \neq 0$, the product must be normalized, i.e., each digit is shifted one to the right, m_n is discarded and $c_n = m_{2n+1}$ is then stored on the address of a_n (or b_n).

The number of memory accesses required is:

$$\text{Memory accesses} = \underbrace{\underbrace{2N^2}_{\text{operand fetches}} + \underbrace{N}_{\text{stores}}}_{\text{multiplication of the mantissas}} + \underbrace{\underbrace{(N-1)}_{\text{fetches}} + \underbrace{N}_{\text{stores}}}_{\text{normalization}}$$

where N = n+1 is the number of hexadecimal digits in each operand. Notice, however, that in the computation of each m_i, one operand fetch may be saved since the operand is already available from the last operation in the previous computation. This saves N-1 operand fetches.

Therefore: Total number of memory accesses = 2N(N+1), including normalization.

Finally, it should be pointed out that the operations may be arranged in such a way that not only (N-1) fetches are saved as described but also each address is modified only in unitary decrements or increments. Since the address registers will have the capability of unitary increment or decrement, only the addresses of a_0 and b_0 are needed initially. These addresses are then possibly indexed and the rest of the multiplication does not require further address broadcasts. Figure 7 illustrates the order of operations for the multiplication of two 4-digit numbers.

3.3 The System as a Whole

A summary description of the complete system is initially presented in order to establish the function of each component and their interconnections. Figure 8 is a diagram of the global structure. The components are:

a) The PU array, containing "a large number" of PU's arranged in rows. Each row has 128 PU's and the number of rows is not fixed: with the exception of "row gating," nothing in the machine is a logical function of the number of PU rows. Therefore, any number of PU rows can be used in SPEAC provided that the row gating contains that same number of inputs. There are, however, some practical limits: too few rows (say 1 or 2) will lead to an

$a_0 b_0$ (initial address broadcasts and fetches)

$\underset{\square}{a_0} \underset{\triangle}{b_1} + \underset{\triangle}{a_1} \underline{b_0}$

$\underset{\square}{b_0} \underset{\triangle}{a_2} + \underset{\triangle}{b_1} \underline{a_1} + \underset{\triangle}{b_2} \underline{a_0}$

$\underset{\square}{a_0} \underset{\triangle}{b_3} + \underset{\triangle}{a_1} \underline{b_2} + \underset{\triangle}{a_2} \underline{b_1} + \underset{\triangle}{a_3} \underline{b_0}$

$\underset{\square}{a_3} \underset{\triangle}{b_1} + \underline{\underset{\triangle}{a_2}} \underline{b_2} + \underline{\underset{\triangle}{a_1}} \underline{b_3}$

$\underset{\square}{b_3} \underset{\triangle}{a_2} + \underline{\underset{\triangle}{b_2}} \underline{a_3}$

$\underset{\square}{a_3} \underset{\triangle}{b_3}$

$\square$ No fetch or address modification

$\triangle$ Add 1 to the address and fetch

$\underline{\quad}$ Subtract 1 from the address and fetch

Figure 7. Fetches in Multiplication

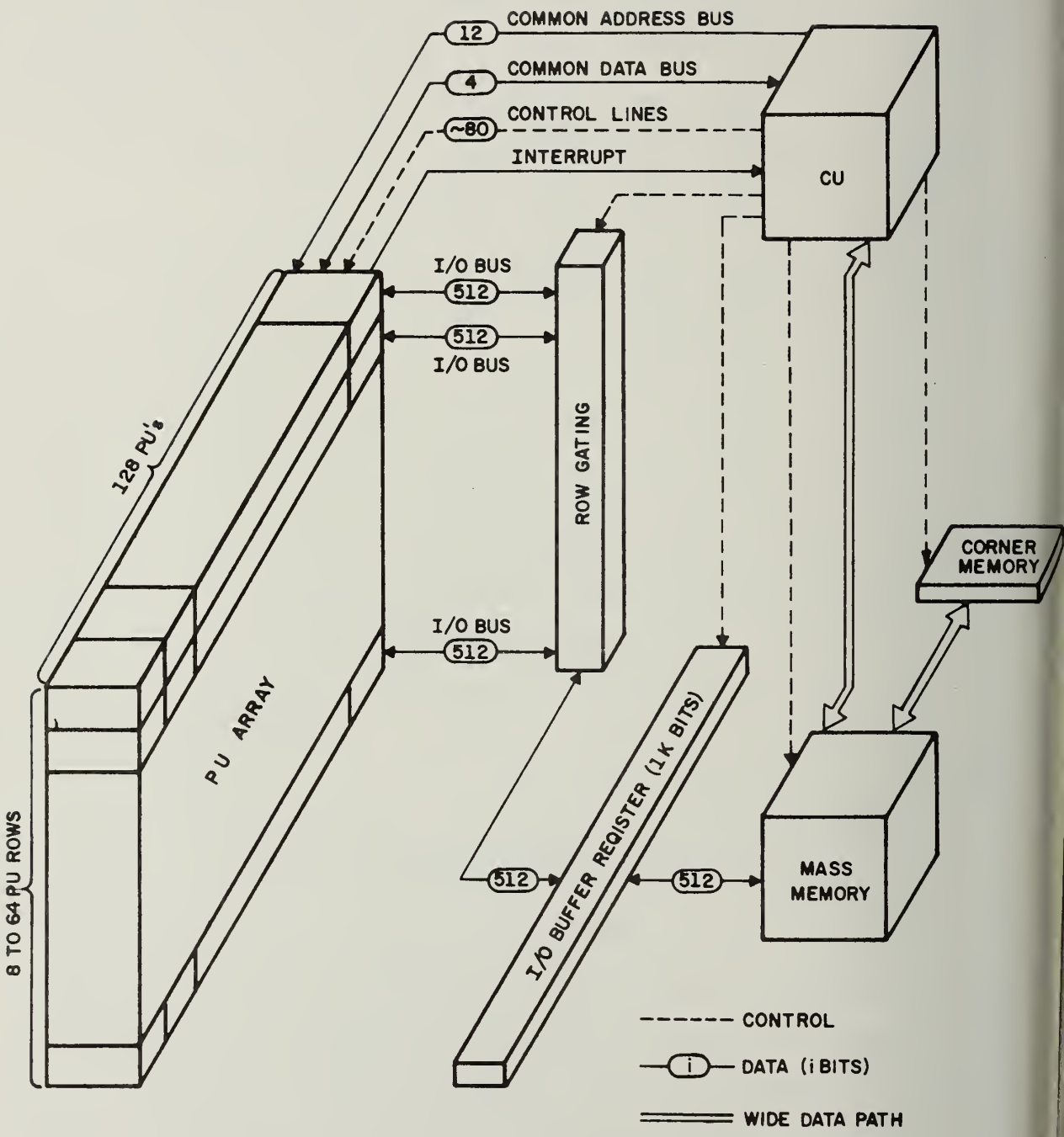


Figure 8. Global Structure

uneconomical machine since each PE is relatively slow and good average speed can only be obtained by using a large number of PE's. Therefore, the speed obtainable with 1 or 2 rows would not justify the investment represented by the components needed to drive the array: CU, mass memory, etc. On the other hand, too many rows will result in poor I/O speed and routing speed (since these operations are performed on a per-row basis) causing a degradation in system performance. Based on these considerations, an interval of 4-64 PU rows has been established as the most useful range. In particular, 8 rows were chosen for the "typical" SPEAC. Therefore, for the remainder of this paper, a 1024 PE machine will be described.

b) The row gating switch which is a 512-bit, bidirectional, 1-out-of-8 selector driven by a row address supplied by the CU. This switch selects one of the PE rows for I/O transactions with the mass memory.

c) The I/O buffer register which is a long, shiftable register to buffer the I/O flow between mass memory and PE array. It should be pointed out that this register has twice the length of the mass-memory word and can be shifted by any multiple of 4-bits in a maximum of 7 clock pulses. These two features enable the I/O buffer register to provide routing facilities for SPEAC. The method will be detailed in Sections 3.7 and 4.7.

d) A mass memory system with at least 10^8 bits of relatively fast (1 to 3 μsec cycle time) random-access memory. Bulk core is the present choice for the mass-memory, probably backed-up by a

hierarchy of large capacity disk and tape. The random-access mass memory serves as a common pool of data for the different parts of the system and is directly accessible to the CU, PU array, corner-memory and other peripherals.

e) A corner-memory which is a special-purpose peripheral device operating on the mass memory in the same fashion as an independent I/O channel. This device is capable of reading from mass memory 128 words with 128 hexadecimal digits each; the $i^{\underline{th}}$ word read can be written as: $a_{i1}\ a_{i2} \cdots a_{i128}$, where each a_{ij} is a hexadecimal digit. After being loaded with rows in this way, the corner-memory can write back in mass memory in a column-wise fashion; i.e., the $i^{\underline{th}}$ word written will be: $a_{1i}\ a_{2i} \cdots a_{128i}$. Therefore, the device can read a matrix of 128×128 hexadecimal digits row-by-row and rewrite the same matrix column-by-column. This function is desirable in SPEAC to convert data written in mass memory by the array into a form that will allow the same data to be easily handled by the CU. The corner-memory is not an essential part of the system but has been included for the sake of completeness. It should also be mentioned that several other peripheral devices (tape decks, printers, etc.) can be attached to the system in the same way as the corner-memory.

f) A control unit (CU) which sends control pulses to all other units in the system besides having full processing capability on its own. Actually, the CU can be considered a standard serial high-speed general purpose computer in which several modifications were introduced. It must accept three different types of

instructions: CU instructions, PE instruction and I/O instructions. CU instructions are completely processed in the CU although operands can be received from the array and results "broadcast" to the array via the common data bus (CDB) which will be described shortly. PE instructions are decoded in the CU and each corresponds to a micro-program which is executed and generates a set of control pulses or micro-sequences. These are sent to every PE in the array via the control lines. Finally, I/O instructions are decoded in the CU and sent to one or more independent I/O channel(s) which drive the row gating, mass memory, I/O buffer register and corner-memory. The CU must also be compatible with the mass memory used in the system since this memory will be shared by the CU and PE and serves as a common pool of data. The CU can interchange data with the PE's via the common data bus, one hexadecimal digit at a time. However, the only high capacity data link between CU and array is via the mass memory. Notice also that SPEAC's programs are not stored in the PEM's but in the CU's own internal fast memory and, for large overlayable programs, also partly in the mass memory.

The control unit is linked to the PE's by three buses and one interrupt wire. The first bus is a 12-bit common address bus (CAB) in the direction of CU to PE only. The CU can send addresses to the array via CAB. These addresses can then be stored by each PE in internal address registers and used to access PEM. The second bus is a 4-bit bidirectional common data bus (CDB) whose use has already been described. The last bus is a set of approximately

80 control lines which control every PE function. The interrupt wire is a single line connecting every PE to the CU. It is used to send to the CU an interrupt request which orginated in a PE and must be serviced by the CU.

Each PE is linked to the row gating by a bidirectional 4-bit I/O bus (IOB) which is not common. All the I/O buses (one from each PE) are connected to the row gating which selects one group of 128 IOB's (corresponding to one PU row) for connection to the I/O buffer register (IOBR).

It is now possible to describe how a program is processed in SPEAC: Program and data are assumed to be initially on tape. The tape is loaded into SPEAC's mass-memory and from there the program is loaded in the CU memory and a portion of the data is transferred to PEM. Processing is then performed simultaneously with further transfers between PEM and mass memory with the latter serving as overlay memory for the relatively small PEM. The results of the computation are transferred from PEM to mass memory and can then be printed or stored in tape via a peripheral device.

Each component of the system will now be analyzed with special emphasis on the PU.

3.4 The Processing Unit

3.4.1 PE Memory

Semiconductor memories were chosen for the PEM's for two basic reasons:

a) Small size, compatible with the LSI chips that make up the PE. This way each PU could be entirely mounted on a single printed circuit card or on a ceramic substrate.

b) Low price per bit even in small size. This characteristic was needed since each PEM in SPEAC is necessarily small for economic

reasons: 8K bits is the proposed basic size with provision for expansion up to a maximum of 32K bits per PEM.

The next step was to choose between bipolar and MOS memories. At the beginning of the investigation, a survey of semiconductor memories [7] indicated that MOS LSI held the greatest potential for this application: large densities (1000 bits per chip is already commercially available), minute power dissipations (50 μw per bit is obtainable), acceptable speeds (less than 1 μsec cycle time is typical) and low price ($.02 per bit is commercially available). Therefore, the following PEM chip was postulated for use in SPEAC: MOS LSI, 1024 bits, 50 μw per bit power dissipation, 500 nsec cycle time, price less than $20 in quantities.

Since progress in the area of semiconductor memories has been so fast, a reevaluation of the design choice for SPEAC's PEM was undertaken at the end of the investigation. It was then discovered that the case for MOS was not as clearcut as before, due to the following factors:

a) Although MOS currently appears to have a distinct density and price advantage, it should be noted that recently announced bipolar processing technology will allow 1024 bit and larger bipolar memories with not much increase in power requirements. These devices will be available for delivery about mid-1972 at about MOS prices. With power reduction techniques they take about the same or less power than MOS and are considerably faster with an 80 to 100 nsec cycle time.

b) It should be noted that the choice of MOS requires an additional power supply level. If bipolar is chosen, the same supply used for the PE logic can be used by PEM. This is more economical

since it is less expensive to buy "x" additional amps on an existing supply than to buy the first "x" amps on a new voltage level.

c) If MOS is used, an interface is normally needed to adjust MOS voltage level to bipolar, thus increasing the number of gates per PE. Moreover the larger densities in MOS are obtainable in dynamic memories; i.e., memories in which the information is stored as charge in MOS P-N junction capacitance. These memories are thus volatile and must be refreshed as often as every 2 μsec at higher temperatures. This is unacceptable in SPEAC since it would introduce frequent delays in processing to refresh PEM. Therefore, static MOS memories must be used and density with these memories is not better than with bipolar. Static MOS is also slower unless decoding is separately performed with bipolar logic.

In conclusion, the factors considered above indicate that PEM would probably be built with bipolar devices or at least static MOS with bipolar decoding if prices drop as much as predicted. In fact a hybrid chip already exists which, if obtainable at a price small enough, would be an excellent choice for PEM: It consists of 8 MOS static memory chips with 256 bits each, mounted on a ceramic pack with bipolar decoding. The organization is 1024 2-bit words making only four of these elements needed for the PEM.

The devices are made by T.I. (SMA 2002) and have a typical cycle time of only 150 nsec. A block diagram is presented in Figure 9.

Therefore, although the basic cycle time of 500 nsec (300 nsec access time) is retained for the remainder of the paper, it now appears that it is a little pessimistic. Significant gains in performance could be obtained in some

operations with the faster memories which would probably be available if SPEAC were to be built in the near future.

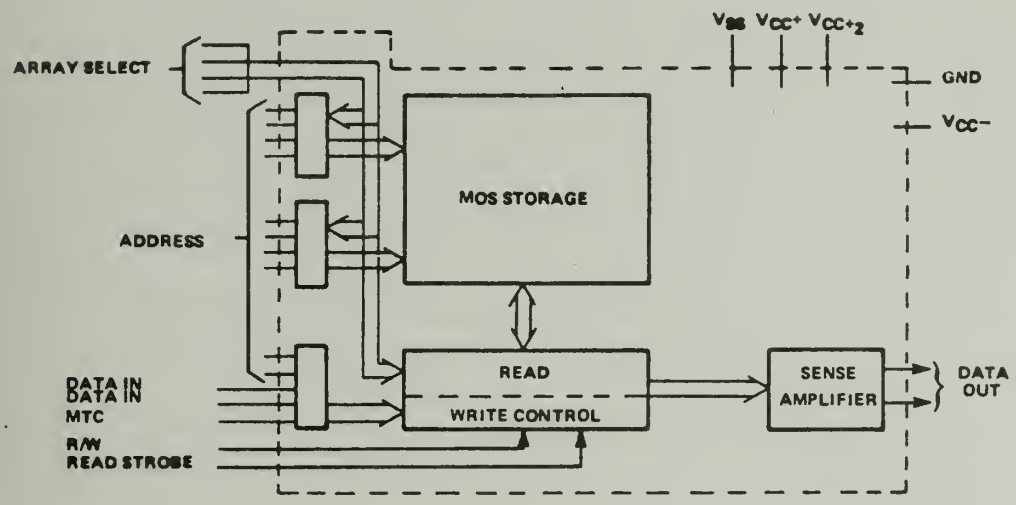


Figure 9. Block Diagram of a Possible PEM Chip

Since the basic unit of data in the PE is one hexadecimal digit, PEM is organized in 4-bit words. Each hexadecimal digit is addressable in the memory. It is also extremely important to adopt an access technique for PEM which will avoid I/O bounding of programs as much as possible: PEM contains only 2K hexadecimal digits or 256 32-bit words. Therefore, for many problems the data will not fit entirely in PEM and mass memory is used as back-up. It would be desirable then to be able to exchange data between PEM and mass memory and, simultaneously, allow the PE to access PEM to perform normal processing. This justifies the adoption of a two-port system: PEM is divided in two modules, each with 1K hexadecimal digits and the two modules can be accessed simultaneously. Basically, one module is replenished from mass memory while the other module is used for operations. In this way, PEM can almost be considered as a fast scratchpad memory for the PE's with mass memory being the main memory.

Since (as will be shown in Sections 3.5 and 4) a row of 1024 32-bit numbers can be exchanged between PEM and mass-memory in about 128 μsec and the basic floating-point operations take on the order of 25 μsec, a number brought to PEM must be used at least six times in operations before being overwritten in order to avoid I/O bounding. This ratio of 1 to 6 is a comfortable figure for a machine intended for scientific applications. It should also be pointed out that I/O-PE overlap is not the only use of the two module system: if I/O is not occurring, the two modules can be used to overlap fetches for CU operations and PE operations or even for the simultaneous fetch of two operands in a PE operation if each operand happens to be in a different module. It is the responsibility of CU's final station (FINST) to assign use of the two PEM modules in an optimum way (see Section 3.5).

3.4.2 PE Data Registers

The algorithm described in Section 3.2 can be very efficiently mechanized using the register structure presented in Figure 10.

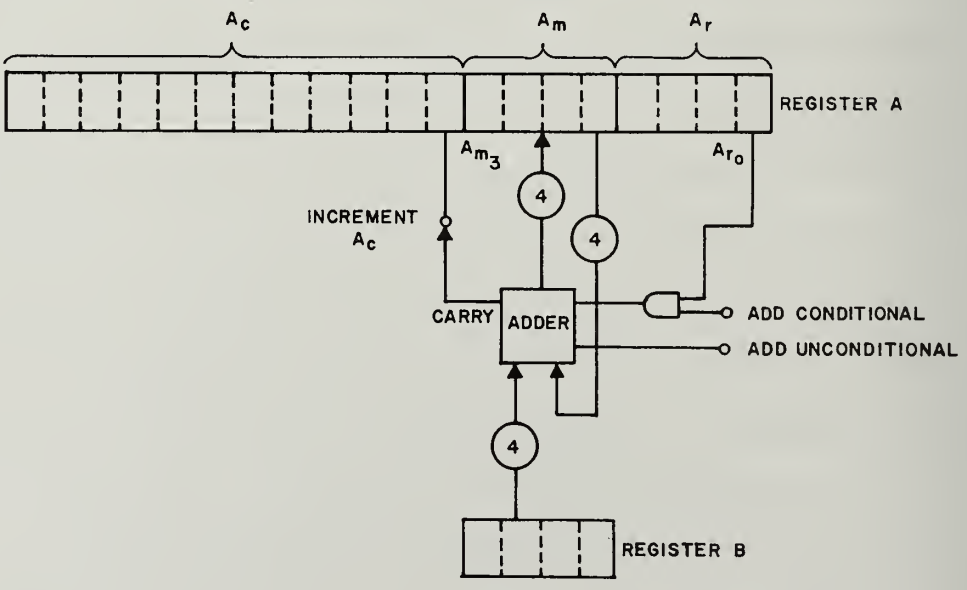


Figure 10. Basic Data Register Structure

There are two data registers: A and B. Register B is a simple, non-shiftable 4-bit unit. Register A is divided into three parts: A_r, A_m (for right and medium) with 4 bits each and A_c (for carry) with 12 bits. Register A is fully shiftable, right or left, bit-by-bit. There is also a fast 4-bit shift mode in which the contents of register A are shifted (left or right) one hexadecimal digit in one operation. The right fast 4-bit shift is not essential to implement the multiplication algorithm efficiently but can be very useful in other applications. It should also be pointed out that part A_c of register A is connected as a counter and a pulse to the "increment A_c" control will cause the contents of A_c to be incremented by one unit. Finally, registers A_m and B are linked by a 4-bit parallel adder which, when activated, replaces the contents of A_m with the sum of the contents of A_m and B. The adder can be used unconditionally or conditioned to the presence of a "one" in location A_{r_0}. The carry generated by the adder can be fed to the "increment A_c" control.

To use the structure of Figure 10 to multiply using the polynomial algorithm, two hexadecimal digits a_i and b_i are placed in registers A_r and B respectively. Multiplication is accomplished with a sequence of four add conditionals and shifts right 1 bit. Register A is then shifted left fast 4 bits and a new multiplication can be performed with the new product automatically added to the previous one(s). Registers A_c and A_m then work as a small accumulator in multiplication. Note that in the polynomial multiplication of two numbers, each n hexadecimal digits long, the worst case carry that can occur is less than $\log_2 n + 4$ bits. Therefore, the number of bits needed in A_c is given by $\log_2 n_{max} + 4$. A reasonable value for n_{max} is 64 which leads to an

A_c 10 bits long. Since in SPEAC register length is naturally a multiple of 4 bits, 12 bits were reserved for A_c. For the same reason, the address register's length was chosen as 12 bits allowing up to 4K hexadecimal digts per PEM module although only 1K is contemplated at this stage.

3.4.3 PE Description

The data register configuration described in the previous section was used as a kernel around which the whole PE was designed. Figure 11 presents a simplified PE diagram showing all registers and data paths. For a complete logical diagram, Figure 13 should be consulted. In order to reduce the size and complexity of Figure 13, a number of special symbols were adopted. These are defined in Figure 12 and deal with representing groups of 4 or 12 wires in a concise way. Only a few logic elements appear explicitly in Figure 13; most logic is represented as logical blocks called packages. These packages are numbered and labeled with a name describing their function; i.e., 1-of-8 selector, type D flip-flop, inverter, etc. The complete diagrams of the logic inside each package are presented in Appendix A. It should be noted that most packages perform standard logic functions and are availabe as SSI or MSI chips. This aspect will be further pursued in the section on implementation.

3.4.3.1 Registers and Buses

Each PE contains nine registers with a total capacity of 65 bits. Table 1 lists each register, its capacity, function, and special features. Buses are used to provide data paths between the different registers. This allows maximum flexibility (since each register can be directly loaded from any other register) at a reasonable cost. Two types of buses are needed: a

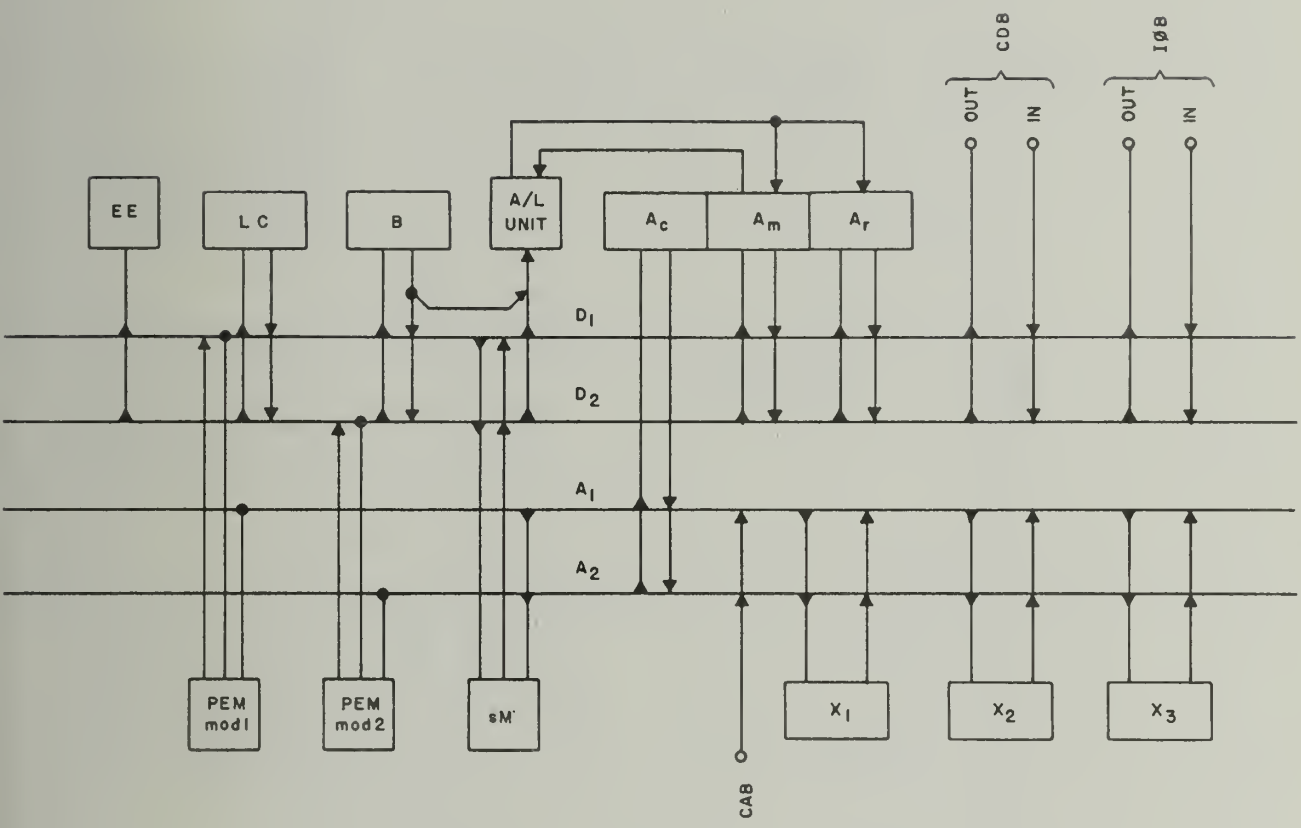


Figure 11. Simplified PE Diagram

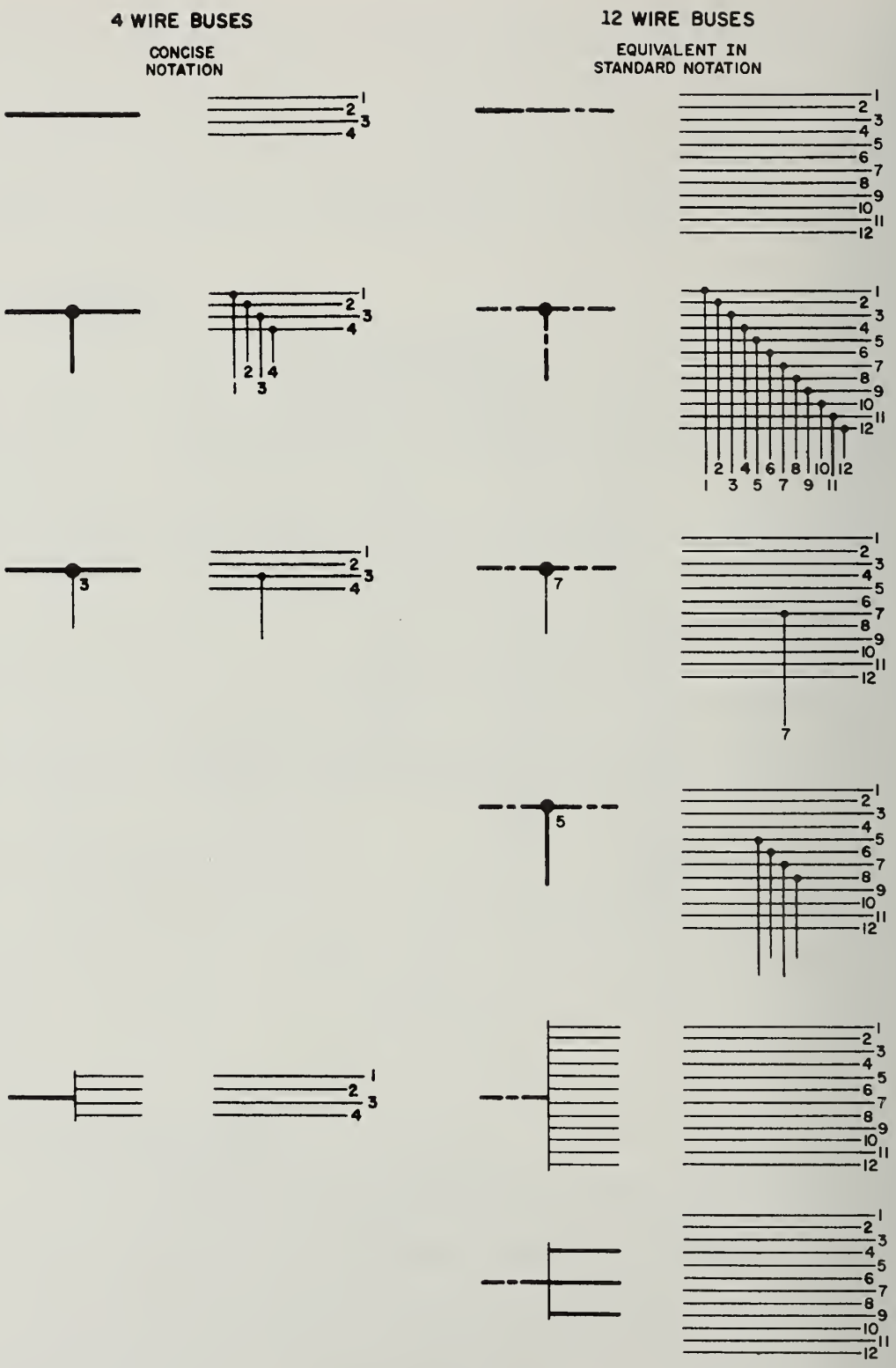


Figure 12. Conventions Used in PE Logical Diagram

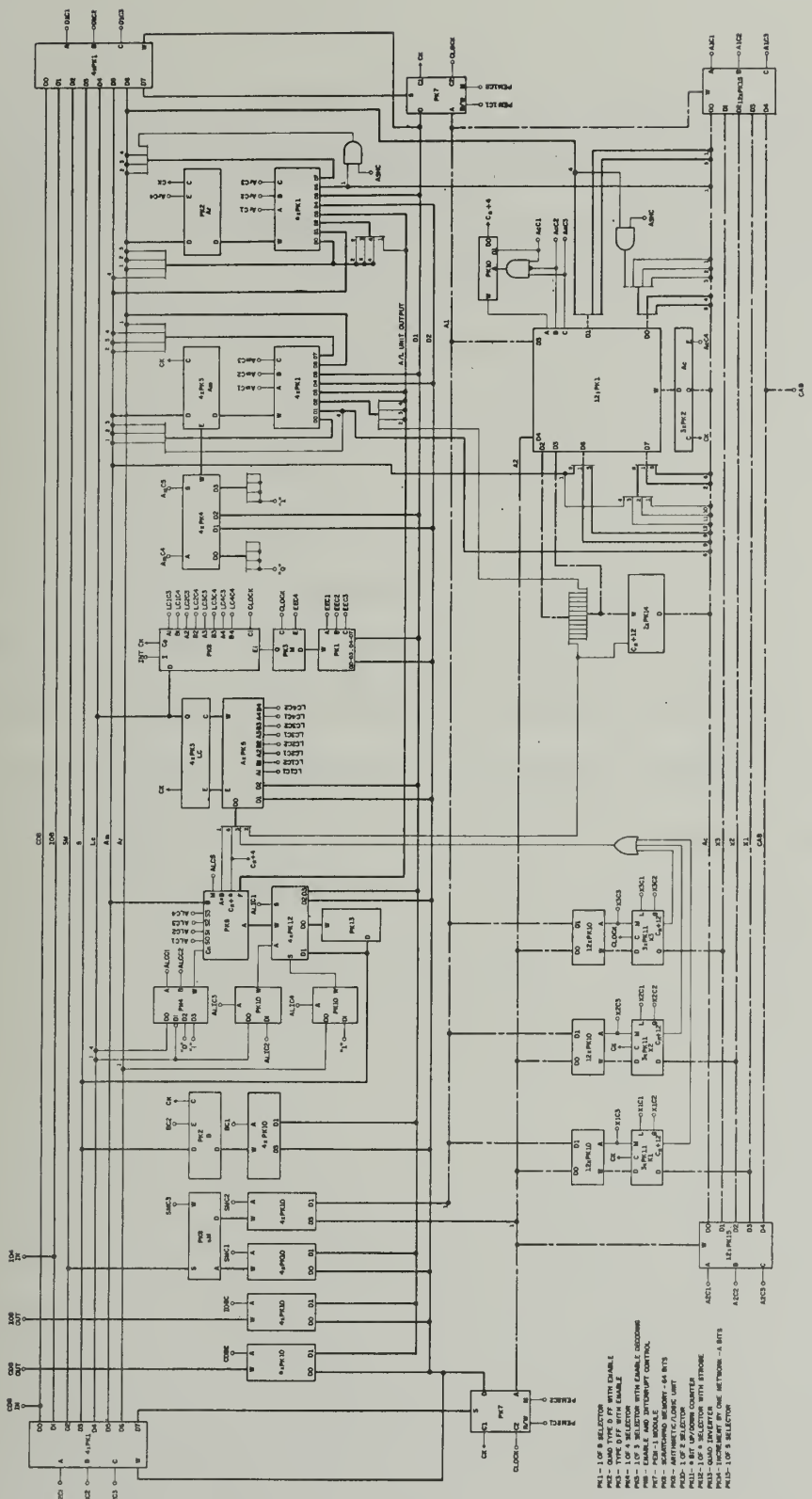


Figure 13. Complete PE Logical Diagram

12-bit address bus, linking all address registers and the CAB, and a 4-bit data bus linking all the remaining registers, the CDB and the IOB. Since it was decided that both PEM modules should be simultaneously accessible, one pair of buses is dedicated to each PEM module. Therefore, there are four buses altogether: two address buses (A1 and A2) and two data buses (D1 and D2). Buses A1 and D1 are linked to PEM module 1 and buses A2 and D2 are linked to PEM module 2. Figure 11 clearly shows all the connections to each bus. In this figure, an arrow into a bus indicates that the given data can be gated into the bus; an arrow out of a bus indicates that the contents of the bus can be gated into the given unit; a dot in the intersection of a wire and a bus indicates a permanent connection of the wire to the bus. It should also be noticed that every line connected to an address bus represented in fact 12 wires (except the line into SM which is a 4-bit line) while lines connected to a data bus stand for 4 wires with the exception of the line into EE which is a single bit line. A very rough approach to the number of gates needed to implement the bus system can now be obtained: counting each arrow associated with a data bus as 4 gates and each arrow associated with an address bus as 12 gates, one obtains a total of 354 gates. This represents about a third of the total number of gates used in the PE with flip-flops accounting for the second third and arithmetic, decoding and local control using the remaining gates.

It is important to point out that PEM module 1 is permanently connected to bus 1 and module 2 to bus 2. Therefore, if an operand is in module i then bus i must be used to fetch that operand. On the other hand, inter-register transfers can use any bus that is available. This fact will be important in the design of the CU's final station (FINST).

Register	Capacity (bits)	Function	Special Features
A			Shiftable (bidirectional, 1- and 4-bit distances)
A_c	12	address/ data	Can count up
A_m	4	data	Each bit is individually enabled
A_r	4	data	None
B	4	data	None
X_1	12	address	Can count up or down
X_2	12	address	Can count up or down
X_3	12	address	Can count up or down
LC	4	local control	Each bit is individually enabled
EE	1	mode	None

Table 1. PE Registers

3.4.3.2 The Arithmetic/Logic Unit

The simple adder of Figure 10 was replaced in the final design by a more sophisticated arithmetic/logic unit (A/L unit) which is capable not only of adding but also of performing several other arithmetic and logic functions as well as comparisons. This unit, whose logical diagram can be seen in package 9 (Appendix A), is currently available from several manufacturers in a 24-pin MSI bipolar chip. There are five control lines in the A/L unit, allowing a choice between 32 functions (not all different). Table 2 shows these 32 functions. There is also an A = B output to test for equality. Other comparisons can be performed by subtracting the two inputs and analyzing the output carry. Input B to the A/L unit is always register A_m. Input A can be selected among D1, D2, reg B and $\overline{\text{reg B}}$. This allows one to compute not only (reg B) - (reg A_m) (by picking reg B as the A input to the A/L unit and subtracting) but also (reg A_m) - (reg B) (by picking $\overline{\text{reg B}}$ as the A input and adding). Inputing to the A/L directly from D1 or D2 is not essential but speeds up several operations by avoiding unnecessary loads into B only to use the A/L. The output of the unit can be gated either into A_m or into A_r. Another important feature is the possibility to gate the output of A/L into A_m shifted one to the right. This speeds up multiplications considerably since two hexadecimal digits can be multiplied in 4 clocks instead of 8 (i.e., 4 add and shift as opposed to 4 adds and 4 shifts).

3.4.3.3 Scratchpad Memory

A small (16 hexadecimal digits), fast scratchpad memory (sM) has been added to the final version of the PE. This unit is available in a 16-pin MSI chip (see package 8, Appendix A) and can read or write one hexadecimal digit in one PE clock. Although not essential to the PE, sM can be added at a

$S_3S_2S_1S_0$	M = 1 (logic functions)	M = 0 (arithmetic operations)	
		$C_n = 0$	$C_n = 1$
0000	$F = \overline{A}$	F = A	F = A + 1
0001	$F = \overline{A \vee B}$	$F = A \vee B$	$F = (A \vee B) + 1$
0010	$F = \overline{A}B$	$F = A \vee \overline{B}$	$F = (A \vee \overline{B}) + 1$
0011	F = 0	F = 1111	F = 0
0100	$F = \overline{AB}$	$F = A + A\overline{B}$	$F = A + A\overline{B} + 1$
0101	$F = \overline{B}$	$F = (A \vee B) + A\overline{B}$	$F = (A \vee B) + A\overline{B} + 1$
0110	$F = A \oplus B$	F = A - B - 1	F = A - B
0111	$F = A\overline{B}$	$F = A\overline{B} - 1$	$F = A\overline{B}$
1000	$F = \overline{A} \vee B$	F = A + AB	F = A + AB + 1
1001	$F = \overline{A \oplus B}$	F = A + B	F = A + B + 1
1010	F = B	$F = (A \vee \overline{B}) + AB$	$F = (A \vee \overline{B}) + AB + 1$
1011	F = AB	F = AB - 1	F = AB
1100	F = 1	F = A + A	F = A + A + 1
1101	$F = A \vee \overline{B}$	$F = (A \vee B) + A$	$F = (A \vee B) + A + 1$
1110	$F = A \vee B$	$F = (A \vee \overline{B}) + A$	$F = (A \vee \overline{B}) + A + 1$
1111	F = A	F = A - 1	F = A

Table 2. Functions Provided by the A/L Unit

low cost and provides a dramatic improvement in performance. Floating-point addition, for example, is speeded up by a factor of three. The main use of sM is to avoid repeated fetches of the same digit in multiplication and to store partial results before normalization. It should be noticed that since sM receives addresses from the address buses (four low order bits only are used), it can be locally indexed, i.e., each PE can locally modify an address in sM before performing an sM fetch. This is extremely valuable in floating-point normalization. Therefore, sM is the fourth element in SPEAC's memory hierarchy which is, from the smallest and fastest unit to the slowest and largest: sM - PEM - mass memory (random access) - large capacity disk.

3.4.3.4 Address Registers

There are three address registers in the PE: X_1, X_2 and X_3. These are simple, non-shiftable 12-bit units with additional logic to enable them to act as up/down counters (see package 11, Appendix A). The address registers are normally loaded from the CAB with a base address broadcast by CU to all PE's. This base address can then be locally indexed. Successive hexadecimal digits of an operand can be accessed by incrementing or decrementing an address register using the up/down counter feature and avoiding frequent use of CAB and repeated local indexing operations. It is now clear that a memory transaction may use as address one of four sources: registers X_1, X_2, X_3, and CAB. The common address bus can be directly used as the address source in I/O transactions or in operand fetches when local indexing is not necessary. This use of CAB indicates that one could possibly eliminate X_3 and still obtain good performance since, in most cases, for PE operations only two addresses are simultaneously needed; in the fetch phase of the operation, the addresses of

the two operands are stored in X_1 and X_2. In writing the result two other addresses are needed in X_1 and X_2--the address of the result and an sM address. X_3 is used, most of the time, to hold I/O transaction addresses. It is felt that eliminating X_3 would cause frequent conflicts in CAB use and a degradation in performance. Only extensive simulation can indicate whether such degradation is small enough to warrant removal of X_3 for a very significant saving in the number of gates.

3.4.3.5 Register A

There are eight possible sources of input data to each of the parts of register A. Six of these eight are common to A_c, A_m and A_r. They are: 1) shift A right one, 2) shift A left one, 3) shift A fast 4 right, 4) shift A fast 4 left, 5) load with D1 (A1 in the case of A_c), and 6) load with D_2 (A_2 in the case of A_c). The seventh input option is the add and shift especially implemented to speed up multiplication. The effect of this input is the following: the output of the A/L unit is loaded into (A_{m_2}, A_{m_1}, A_{m_0}, A_{r_3}), A_r is shifted right one and A_c is either shifted right one(if the output carry for the A/L unit is zero) or is incremented by one and shifted right one (if the output carry from the A/L unit is one). Finally, the eighth and final possible input to A is: for A_m and A_r, the output of the A/L unit (used for addition, subtraction and logical operations); for A_c, the last input possibility is simply A_c incremented by one (i.e., the counter feature of A_c).

Input control is independent for each of the three parts of register A. Therefore, register A shifts end-around as a whole only when A_c, A_m and A_r are simultaneously loaded with the same shift input. Several other useful results may be obtained when only one or two of the parts of A receives a shift

command. For example, loading A_r with a shift fast 4 right enables one to copy A_m directly into A_r without having to use D1 or D2. A direct swap of the contents of A_m and A_r can be achieved by simultaneously loading A_m with a shift fast 4 left and A_r with a shift fast 4 right. There is a control wire to determine whether a distance 1 shift is to be end-off or end-around. Distance 4 end-off shifts are obtained by shifting only two of the parts of A.

A_c and A_r have a single load control which, when OFF, preserves the contents of the register and when ON loads the register with the selected input. Load control for A_m is more sophisticated and allows not only "load" and "no-load" but also a conditional load dependent on the value in D1 or D2. In this conditional load, bit i of A_m is loaded only if bit i in D1 or D2 is ON. This is very useful in "assembling" a hexadecimal digit out of specific bits of two other digits as is the case in inserting a sign bit in a number.

It is important to notice that A1 or A2 can be gated into A_c thus allowing addresses to reach the data handling part of the PE. This feature is used to modify addresses in local indexing. Also A_c is a counter and can be used as such when not needed to accumulate a carry in multiplication. This provides a general purpose 12-bit counter in the PE which is extremely useful in several applications. Therefore, A_c has a quadruple function: a) it provides linkage between the address portion and the data portion of the PE, b) it serves as a general purpose counter, c) it accumulates the carry in multiplication, d) for special applications, A_c could be used as an additional address register.

For a more complete idea of the whole PE as well as all the available controls the reader is directed to Figure 13 where each control wire is indicated as a line ending in an open circle with a code name associated to it. There is

Control Wire	Controls	Function
AcC1 to AcC3	A_c	Select one out of eight possible inputs
AcC4	A_c	Load A_c with the selected input
ALCC1, ALCC2	A/L unit	Select input carry C_n between 0, 1, $\overline{\text{lcFF1}}$ and lcFF4
ALC1 to ALC5	A/L unit	Select function performed by A/L unit (see Table 2)
ALIC1, ALIC2	A/L unit	Select operand A for A/L unit between B, $\overline{B}$, D1 and D2
ALIC3	A/L unit	Use $\overline{B}$ instead of B as operand A if lcFF1 is ON
ALIC4	A/L unit	Uses 0 instead of selected data as operand A if A_{r_0} is OFF
AmC1 to AmC3	A_m	Select one out of eight possible inputs
AmC4, AmC5	A_m	00 - do not load A_m; 11 - load A_m (all bits) with selected input; 10 - load A_m with AND of selected input and D1; 01 - load A_m with AND of selected input and D2
ArC1 to ArC3	A_r	Select one out of eight possible inputs
ArC4	A_r	Load A_r with selected input
AShC	A	Distance 1 shift is end-around
A1C1 to A1C3	A1	Select one value out of five to gate into A1
A2C1 to A2C3	A2	Select one value out of five to gate into A2
BC1	B	Select among D1 and D2 as inputs to B
BC2	B	Load B with the selected input
CDBC	CDB	Select between D1 and D2 to gate into CDB
Clock	All FF's	Clock pulse
D1C1 to D1C3	D1	Select one value out of eight to gate into D1
D2C1 to D2C3	D2	Select one value out of eight to gate into D2

Table 3. Control Wires and Their Functions

Control Wire	Controls	Function
EEC1 to EEC3	EE	Select one bit out of the eight in D1, D2 as input to EE
EEC4	EE	Load EE with the selected input bit
IOBC	IOB	Select between D1 and D2 to gate into IOB
LCiC1, LCiC2 (i=1,2,3,4)	lcFFi	00 - do not load lcFFi; 11 - load lcFFi with bit i of D1; 10 - load lcFFi with bit i of D2; 01 - load lcFFi with: i=1, A=B output from A/L unit; i=2, output carry from A_c; i=3, OR of carry from X_1, X_2 and X_3; i=4, output carry from A/L unit
LCiC3, LCiC4 (i=1,2,3,4)	lcFFi	00 - do nothing; 10 - gate lcFFi into interrupt wire; 01 - enable clock if lcFFi of OFF; 11 - enable clock if lcFFi is ON
PEMiC1 (i=1,2)	PEM mod i	Select read or write
PEMiC2 (i=1,2)	PEM mod i	Do not obey mode control
sMC1	sM	Select between D1 and D2 as input to be read into sM
sMC2	sM	Select between four low order bits of A1 and A2 as address to sM
sMC3	sM	Select read or write in sM
XiC1 (i=1,2,3)	X_i	Load input selected by XiC3
XiC2 (i=1,2,3)	X_i	Count X_i up or down as selected by XiC3
XiC3 (i=1,2,3)	X_i	If counting, select between up or down; if loading select input between A1 and A2

Table 3 (Continued)

a total of 78 control wires in the PE and Table 3 lists these wires in alphabetical order along with a description of their function.

3.4.4 Local Control

It has already been pointed out that a certain minimum amount of local control must be present at each PE to take care of data-dependent actions. This takes the form of gates which, when activated, allow or inhibit an attempted action depending on some internal PE state. When the information used for local control is stored at some PE register at the same time it is needed, no additional memory elements are necessary. This is the case, for example, with the use of A_{r_0} as local control for the "add conditional" in multiplication (see Figure 10). In other instances, however, the local control information is not available any more when it is needed. In this case local control flip-flops must be introduced to store this information. Specifically, there are in the PE six "dynamic outputs" which must be stored somehow since they may be needed for local control. These dynamic outputs are:

Equality output (A = B) from the A/L unit

Carry (C_{n+12}) from the A_c counter

Carry/borrow (C_{n+12}) from the address registers X_1, X_2 and X_3

Output carry (C_{n+4}) from the A/L unit

Four local control flip-flops designated by lcFFi (i=1,2,3,4) are used to store the dynamic outputs: A = B can be stored in lcFF1; C_{n+12} from A_c can be stored in lcFF2; the OR of C_{n+12} from X_1, X_2 and X_3 can be stored in lcFF3; and C_{n+4} can be stored in lcFF4. Notice that only one lcFF is used to store the OR of the carry/borrow's from the three address registers. This results in a saving of two lcFF's and does not introduce any serious disadvantage

since a carry/borrow in an address register is normally an error condition and will cause an interrupt regardless of the particular register in which the overflow occurred.

It is easy to see that local control is the most serious obstacle in achieving the goal of a PE as general as possible, able to cope with a wide range of word formats and instructions. Normally, a lcFF may be loaded only with a specific bit of information and a certain PE function. This tends to freeze conventions like negative number representation and sign bit location. These shortcomings suggest the possibility of some generalized local control logic as illustrated in Figure 14. This could be viewed as allowing micro-programming at the PE level. Obviously, a generalized local control as the one proposed in Figure 14 is prohibitively expensive. Therefore, the subject was intensively researched and a satisfactory compromise has been found.

Initially, one should notice that any type of local control can be achieved using only enable control; i.e., being able to enable or disable the whole PE according to the presence of a ZERO or a ONE in a lcFF. To prove this proposition, simply consider the fact that local control can be of two types: a) if (lcFFi) THEN action 1, and b) IF (lcFFi) THEN action 1 ELSE action 2. For the moment, a disabled PE is defined as one in which the clock is inhibited causing all registers to retain their old values. Local control of type a can be implemented by enabling only the PE's in which lcFFi is ON, executing the microsequence to perform action 1 and then enabling all PE's again. For local control of type b a second step is needed in which only PE's in which lcFFi is OFF are enabled and then action 2 is executed followed by enabling all PE's again. This type of local control, achieved through enabling and disabling PE's, will be called indirect local control as opposed to direct local control in

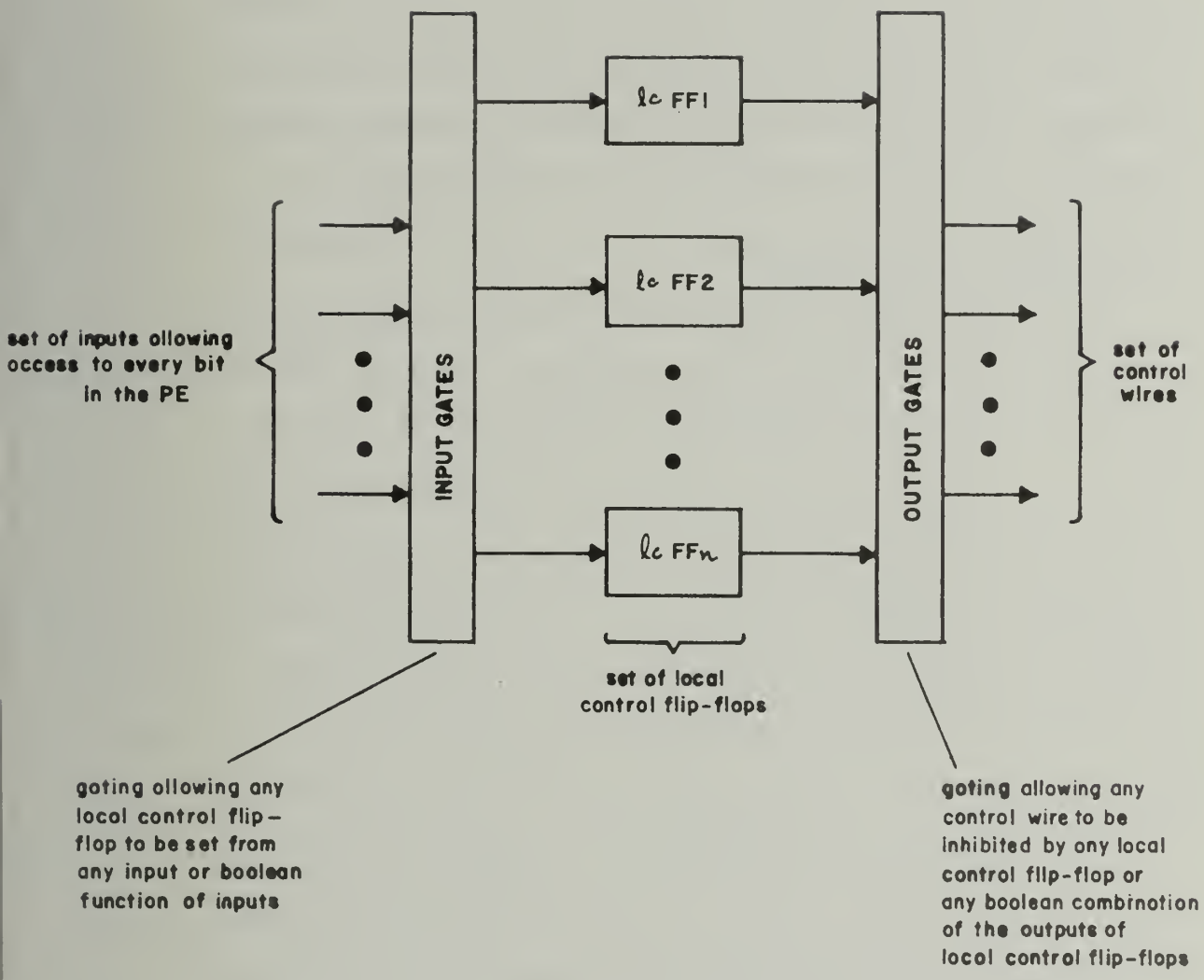


Figure 14. A Generalized Local Control

which one or more control wires are directly inhibited by some lcFF or other register in the PE. Although indirect lc is universal and can achieve any desired effect, it is obviously slower since extra time is needed to turn PE's ON and OFF. Therefore, local control in SPEAC will be primarily of the indirect type except for a few extremely important functions in which one cannot afford the extra time; these will be implemented directly.

3.4.4.1 Direct Local Control

Direct local control is used in SPEAC for four functions:

a) Input carry (C_n) to the A/L unit. This is controlled by wires ALCC1 and ALCC2 (see Figures 13 and Table 3). C_n can thus be chosen between four values: ONE, ZERO, the complement of lcFF1, and the same value as in lcFF4. C_n = ZERO is used in initiating unsigned addition and C_n = ONE in initiating unsigned subtraction (using also $\overline{\text{reg B}}$ as operand A to the A/L unit). Signed addition must be locally controlled since it can be an actual addition (if both operands have the same sign) or a subtraction (if the signs are different). A sign comparison can easily be stored in lcFF1 since A = B can be stored in this flip-flop. Therefore, lcFF1 = ONE if signs are equal, ZERO otherwise and C_n = $\overline{\text{lcFF1}}$ can be used in initiating a signed addition. The last possible value of C_n is lcFF4. This is used in the middle of an addition or subtraction, when C_n must have the value that C_{n+4} had in the previous step. Therefore, when adding (or subtracting) hexadecimal digits a_i and b_i of A and B, the value of lcFF4 is the carry C_{n+4} from the addition (or subtraction) of

a_{i-1} and b_{i-1} and will be used as C_n. At the same time, lcFF4 will be changed to C_{n+4} from $a_i + b_i$, to be used in the next step.

b) Input A to the A/L unit. This is controlled by wires ALIC1, ALIC2, and ALIC3. The first two wires choose between B, $\overline{B}$, D1 and D2. The last one, ALIC3 implements a direct local control; when ALIC3 is ON, input A to the A/L unit will be $\overline{B}$ instead of B if lcFF1 is OFF. If lcFF1 contains a comparison of signs in signed addition, as explained above, then this local control transforms an addition into a subtraction for the PE's in which the signs are unequal.

c) Gating of input A to the A/L unit. This local control is actuated by a ONE in wire ALIC4. When this happens, the gating of input A to the A/L unit is inhibited by the presence of a ZERO in A_{r_0}. Therefore, if A_{r_0} is ZERO and ALIC4 is ON, operand A to the A/L unit is ZERO regardless of the values of ALIC1, ALIC2 and ALIC3. Obviously, this implements the "add conditional" needed for multiplication.

d) Finally, there is local control built into the input gating to register A_c. When "add and shift" is chosen as the input to register A, A_c is either shifted right one (if C_{n+4} is ZERO) or is incremented by one and shifted right one (if C_{n+4} is ONE) as explained in Section 3.4.3.5.

3.4.4.2 Indirect Local Control

All control functions not directly implemented are obtained

using the lcFF's to enable chosen PE's. In order to do this, one must be able to store the controlling bit in one of the lcFF's. It has already been explained that the "dynamic outputs" can be directly stored in lcFF's. There are four lcFF's in the PE and Figure 15 presents a simplified diagram of the controls at the input and output of each lcFF. For the precise logic, the reader is referred to Figure 13 and package 6 in Appendix A.

The local control structure illustrated in Figure 15 is actually a simplification of the generalized local control described in Figure 14; the number of gates was considerably reduced to make the unit practical for use in a "small" PE like SPEAC's. Nevertheless, the unit is as powerful as the generalized local control although not as fast.

In order to perform indirect local control, every bit in the PE should be accessible to a lcFF. This is achieved by linking LC, the register composed of the four lcFF's, to data buses D1 and D2 like all other data registers thus allowing any bit in the PE to be fed as input to a lcFF. It should also be recalled that the dynamic outputs can also be stored in the lcFF's. Therefore, the input gates of Figure 14 have been reduced in Figure 15 to a 1-out-of-3 selector for each lcFF. The selector for lcFFi is controlled by two wires: LCiC1 and LCiC2. The four possible input actions are: a) do nothing (i.e., retain the previous value stored), b) store in lcFFi the $i^{\underline{th}}$ bit in D1, c) store in lcFFi the $i^{\underline{th}}$ bit in D2, and d) store in lcFFi the dynamic output associated with that flip-flop as described in Section 2.4.4.

It is often necessary to set a lcFF to a Boolean combination of other bits, sometimes to a Boolean combination of bits in other lcFF's. In order to save the gates needed to implement this directly, the output of LC is

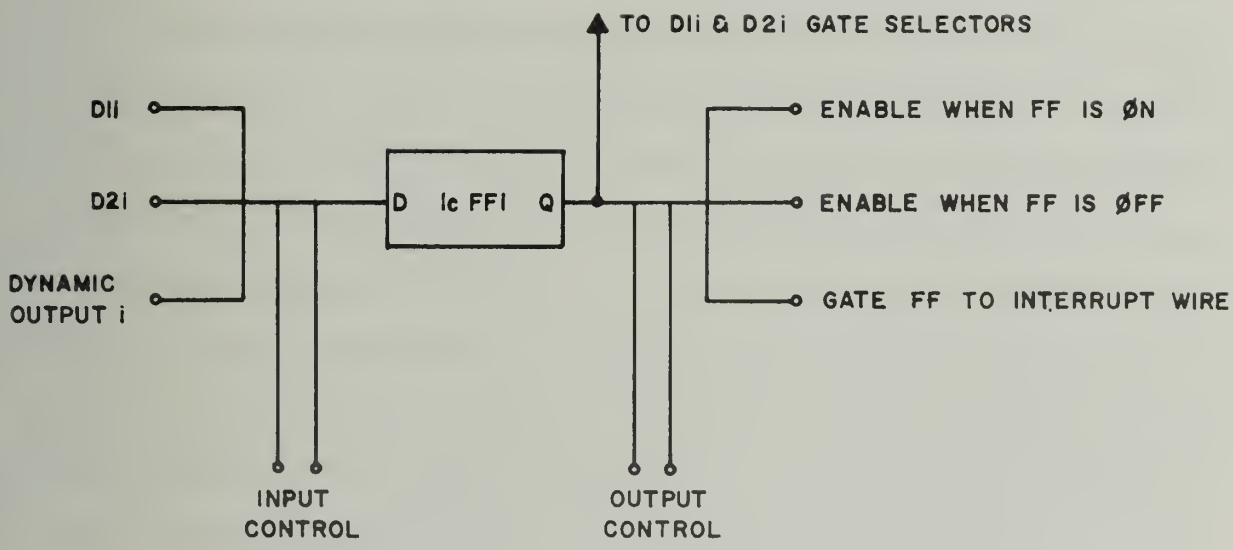


Figure 15. Diagram of a Local Control FF

made available as a possible value of D1 or D2 like any other data register. Therefore, the contents of LC can be brought to register A and one can perform shifts and logical operations. When the desired function is obtained, it can be stored back in LC from D1 or D2.

The output gates of the generalized local control have also been reduced in Figure 15 to a 1-out-of-3 selector controlled by two wires: LCiC3 and LCiC4. These wires control the function performed by each lcFF. The four possible functions performed by lcFFi are: a) do nothing (i.e., the state of the flip-flop has no effect on the PE), b) enable PE only if lcFFi is ON, c) enable PE only if lcFFi is OFF, and d) gate the output of lcFFi to the interrupt wire. Function d, used when it is desired to send an interrupt sign to the CU, will be discussed in Section 3.4.6. Functions b and c are used to perform indirect local control. Since it is possible to enable either on a ONE or on a ZERO of a lcFF, one avoids moving LC to A only for complementing. This is important because it is often needed to enable PE's in which lcFFi is ON, perform an action and then enable only PE's in which it is OFF to perform another action thus obtaining control of the type IF (lcFFi) THEN action 1 ELSE action 2. It is then clear that a lcFF does not have a certain fixed function but is attributed, for each clock cycle, one among four possible functions. Also, each lcFF is controlled completely independently from the others, which makes this type of lc rather costly in terms of control wires; 16 wires are required altogether. It is felt, however, that the performance and versatility obtainable with this local control justifies the cost.

3.4.5 Mode Control

Mode control is simply the ON-OFF type as in ILLIAC IV. Register M

(also called EE for external enable) is in charge of this control. This is a single bit register which can be loaded with any bit of D1 or D2. Therefore, the input gating for register M is a 1-out-of-8 selector controlled by wires EEC1, EEC2 and EEC3. A fourth wire (EEC4) completes the control of register M. When EEC4 is ON, M is loaded with the input bit select by the three other wires; when it is OFF, M retains its old value. The mode control register has a fixed function which is to enable the PE on a ONE (i.e., whenever M is ON, the PE is enabled and whenever it is OFF, the PE is disabled).

The mode register can also be called "external enable" register, which points out the fact that it is an enable register reserved for user (or macro-instructions) manipulations, as opposed to the internal enable, which is the function attributable to lcFF's. This is normally used only by the systems programmer in micro-instructions.

It is now convenient to define precisely what is meant by a disabled PE. Most registers in the PE are clocked by the signal Ck which is the main clock sent by the CU "Clock", inhibited by register M, and possibly by the lcFF's. Therefore, when a PE is disabled, all registers clocked by Ck are frozen; i.e., they retain their old values. The elements not clocked by Ck are: Registers M and X_3, and the two PEM modules. Register M is directly clocked by "Clock" and cannot be disabled. This is obviously needed or else, once M were disabled, the PE could never be enabled again. There is a special problem with PEM and X_3: as described in Section 3.4.1, one must be able to overlap PE operation with replenishment of PEM. Therefore, I/O operations must be able to reach a disabled PE since PEM in all PE's must be replenished regardless of the fact that some PE's may be temporarily OFF. In order to accomplish this, each PEM module receives both clock signals: the direct signal

"Clock" and the possibly inhibited Ck. A control wire (PEMiC2 where i is the module number) decides whether "Clock" or Ck is to be used, thus choosing between ignoring and respecting disabling. Also, X_3 is clocked by "Clock" instead of Ck since it is mainly used to hold addresses for I/O operations.

Finally, it can be pointed out that the contents of M are not accessible to the PE. Therefore, if the setting of M is to be used later, it must be temporarily stored in sM at the time it is being loaded into M.

3.4.6 Interrupts

The interrupt system is very simple; every PE has one interrupt wire and the CU receives also only one wire which is the OR of the data in the interrupt wires of each PE. If one or more PE's are interrupted, the CU will sense a "1" in the interrupt wire and the operating system will have to interrogate the PE's to find out which are responsible. This scheme has the advantage of making the number of interrupt wires independent of the number of PE's, allowing for system expansion.

It has already been described (in Section 3.4.4.2) that one of the functions attributable to each lcFF is the gating of its contents into the interrupt wire. Conditions that should case an interrupt are detected in the PE and stored as a ONE in some lcFF. The interrupt can then be sent to the CU by attributing the interrupt function to that lcFF. It should be noticed that the propagation times of the PE interrupt signals are assumed short compared to the PE clock period. This is what allows only one interrupt flip-flop to be used for different conditions like the following: exponent overflow, exponent underflow, fixed point overflow, division by zero, etc. It is assumed that the CU will notice the interrupt soon enough to be able to distinguish the different conditions by an analysis of which step of which operation

was being performed.

It is also interesting to point out that the interrupt system is used not only to detect error conditions, but can be very useful to detect the end of a recurrence process or to optimize certain programs. For example, assume that a recurrence process is being executed by all PE's. At the end of each step, the error is computed and compared with the maximum acceptable. All PE's in which the error is smaller than the maximum are turned OFF, via lcFF3 for example. Sending lcFF3 via the interrupt wire will enable the CU to detect if all PE's have been turned OFF. If this is the case, the recurrence is ended. It may also be quite useful to add a control wire enabling one to send M via the interrupt wire.

3.4.7 Implementation Remarks

This section considers some of the design problems that would have to be solved if the PE previously described were to be actually built. T^2L integrated circuits will be used in the implementation of the PE logic due to their medium cost, speed, and power dissipation. MOS logic was initially considered and it offered considerable advantages in cost and power dissipation, however, it does not seem to be fast enough for the purpose of making the memory cycle (1/2 μsec) the basic speed limiting factor. This cannot be achieved with conventional MOS logic in the PEM (although silicon-on-saphire technology promises for the near future an order of magnitude increase in the speed of MOS logic). T^2L, although not as fast and desirable, will allow a good balance between memory fetch time and PE operation time; assuming 10 nsec as the typical gate propagation delay time, and considering that there are no long logic chains in the PE, it is realistic to assume a PE clock period of 100 nsec (PE clock frequency = 10 Mc/s). Therefore, a PE clock takes $\frac{1}{5}$ to $\frac{1}{2}$ of a PEM

cycle, depending on how fast the PEM is used.

Since the PU's will be pluggable, it is important that the number of connections to each PU be minimized as this is, in integrated circuitry, a cost factor probably more important than mere gate count. Table 4 shows the actual number of PE connections achieved. A total of 103 to 110 is needed, probably making necessary two connectors in each PU if a conventional printed circuit is used. Three power wires are needed instead of two if MOS PEM's are used since they need an extra voltage level. IOB and CAB must be bidirectional. This is achieved either running two independent buses, one in and one out as indicated in Figure 11 and 13, or using only one bus with additional logic in the PE and one extra control wire to choose in which direction the bus is to be used. The cost of six extra connections seems small enough to save the extra complications of using only one bus. Also, if both in and out buses are present, they could be simultaneously used in some operations like I/O and routing. Therefore, eight wires are used for CDB and eight more for IOB.

Function	Number of Connections
Control wires	80 - 78
CDB	4 - 8
IOB	4 - 8
CAB	12
Interrupt Wire	1
Power	2 - 3
Total	103 - 110

Table 4. Connections to Each PU

The number of control wires (78) is quite large, but this is the price to pay for retaining maximum PE versatility for the micro-programmer. Of course, the number of control wires could be reduced by adding encoding logic in each PE. However, this would increase the gate count per PE and reduce the flexibility of the controls. Therefore, encoding of control wires was used only when flexibility was not affected (like in the input to a register; anyhow, the register cannot be loaded with two different inputs) and when the extra gating comes automatically in the IC's used or can be added economically.

T^2L MSI chips manufactured by Texas Instruments provide a preliminary guideline in the discussion of questions related to: number of gates, IC's available off-the-shelf, power dissipation, etc. Therefore, the suggested IC's are limited to the ones listed in [8] and this information is only useful in rough evaluations for a breadboard PE. In actual construction, a few made-to-order LSI IC's would be used in place of several smaller chips. Table 5 lists a few MSI T^2L chips available off-the-shelf that could be of interest in the construction of a breadboard PE. Table 6 lists all the packages used in Figure 13 and also gives the number of FF's per package and a very rough evaluation of the number of equivalent gates per package. Memory elements were not included in the evaluation of the totals for the PE. Roughly, the proposed implementation requires 1K gates and 64 type D flip-flops for a total of approximately 1.3K gates. Table 7 presents a preliminary evaluation of the number of IC chips that would be needed in each PU. Two numbers are given: one, for a breadboard PU, uses the chips introduced in Table 5; in this case more than one hundred chips are necessary. The second number assumes the availability of a few custom made IC's with up to 24 pins

Chip Number	Type	Equivalent Gates	DIP Pins	Average power diss mW	Description
1	SMA2002	na	28	1331	Memory: MOS, 1024 × 2, T^2L compatible fully decoded
2	Fair3532	na	16	150	Memory: MOS, 512 × 2, T^2L compatible fully decoded
3	SN7489	na	16	375	Memory: 16 × 4, scratchpad
4	SN74175	na	16	na	Register: D-type, 4 bits
5	SN74174	na	16	na	Register: D-type, 6 bits
6	SN74191	58	16	325	Counter: parallel in/out, synchronized, up/down, 4 bits
7	SN74181	75	24	~375	A/L unit: 4 bits
8	SN74157	~15	16	125	Data selector: Quad 2-to-1
9	SN74153	~16	16	180	Data selector: Dual 4-to-1 with strobe
10	SN74152	~15	16	130	Data selector: 8-to-1
11	SN74L98	~40	16	25	Data selector/storage register: 2-to-1, 4 bits
12	SN74LS83	~42	16	75	4-bit binary full adder

Table 5. Some IC Chips that Might Be Used in the PE

Package Number	Function	No. Used	Approx. Gates per package	FF's per package	Total gates	Total FF's
1	1-out-of-8 selector; no strobe	29	9	0	261	0
2	Quad D type FF; clock enabled for all FF's simultaneously	5	5	4	25	20
3	Type D FF with enable on the clock	9	2	1	18	9
4	1-out-of-4 selector; no strobe	5	5	0	25	0
5	1-out-of-3 selector with enable decoding	4	5	0	20	0
6	Enable and interrupt control	1	18	0	18	0
7	PEM-1 mod	2	---	---	---	---
8	sM--64 bit memory--16 4-bit words	1	---	---	---	---
9	A/L unit	1	~60	0	60	0
10	1-out-of-2 selector without strobe	59	3	0	187	0
11	4 bit add/subtract counter, parallel in/parallel out	9	~25	4	225	36
12	1-out-of-4 selector with strobe	4	5	0	20	0
13	Quad inverter	1	4	0	4	0
14	Increment by 1 network (6 bits)	2	25	0	50	0
15	1-of-5 selector	24	6	0	144	0
TOTALS					1057	65

Table 6. Packages Used in the PE and Their Contents

Used In	Breadboard		Actual Implementation	
	Chips Used	No. of Chips	Chips Used	No. of Chips
PEM	chip 1 = $\frac{1}{2}$ Pk 7	4	as in breadboard	4
reg B and input gates	chip 11 as Pk 2 + (4 × Pk 10)	1	as in breadboard	1
input to D1, D2, A_m, A_r, A_c	chip 10 as Pk 1	28	2 × Pk 1	14
input to A_1, A_2	chip 10 as Pk 15	24	2 × Pk 15	12
output to IOB, CDB	chip 8 as 4 × Pk 10	2	as in breadboard	2
inputs to sM	chip 8 as 4 × Pk 10	2	as in breadboard	2
sM	chip 3 as Pk 8	1	as in breadboard	1
A/L unit	chip 7 as Pk 9	1	as in breadboard	1
X_1, X_2, X_3	chip 6 as Pk 11	9	$1\frac{1}{2}$ Pk 11	6
inputs to X_1, X_2, X_3	chip 8 as 4 × Pk 10	9	6 × Pk 10	6
input A to A/L	chip 9 as 2 × Pk 12	2	as in breadboard	2
Increment net	chip 12 as $\frac{2}{3}$ × Pk 14	3	Pk 14	2
A_c	chip 5 as $1\frac{1}{2}$ × Pk 2	2	as in breadboard	2
A_r	chip 4 as Pk 2	1	as in breadboard	1
A_m	SSI dual FF	2	4 × Pk 3	1
enable control in A_m	chip 9 as 2 × Pk 4	2	4 × Pk 4	1
M and M input	SSI FF; chip 10 as Pk 1	2	Pk 3 + Pk 1	1
LC and LC input	SSI dual FF; chip 9 as Pk 5	4	2 × Pk 3 + 2 × Pk 5	2
enable control	chip 9 as $\frac{1}{4}$ × Pk 6	4	Pk 6	1
others	SSI chips	5	Pk 13 + Pk 4; 3 × Pk 5	2
	Total	108	Total	64

Table 7. Rough Estimates for the Number of Chips Per PU

per DIP. These IC's are only slight modifications of the ones in Table 5. In this case, the number of chips goes down to about 64. This number of chips will readily fit in one printed circuit board or, better yet, a new packaging technology could be used: a multi-chip on a ceramic substrate technique which is being developed at Fairchild. As far as design is concerned, the substrate is analogous to a two-sided printed circuit board with single devices installed. In addition, a system package is being developed to connect these devices together with simple cam-operated connectors and backplanes.

It is important to point out that the number of 64 chips was obtained with a very superficial analysis of the circuit and only assuming the availability of quasi-standard IC's. It is expected that with careful computer analysis of the possible partitions of the circuit and wide use of custom-made IC's, the number of MSI chips could go down to about 30 (this is the number reached if one divides the total number of equivalent gates in the PE (1.3K) by 60 to 70, the number of equivalent gates easily obtained nowadays in one MSI T^2L chip).

The power dissipation per PE is quite acceptable. It is on the order of 15 watts, assuming an average of 10 mw per gate for T^2L. A new low power T^2L could be used to reduce this number by a factor of 5 to 10.

Finally, it should be mentioned that a number of simplifications could be adopted in the PE at a small cost in performance. Only careful simulation can decide whether the saving thus obtained justifies the loss in performance or versatility. Some of these simplifications are:

- make B unavailable as a value to D1 or D2
- do not use X_3
- use only 10 bits in address lines instead of 12

- make X_i count up only instead of up/down.
- reduce A_c to 8 or 10 bits.

3.5 The Control Unit

The control unit has already been summarily described in Section 3.3. In this section, a few more details of CU's structure and functions are presented but only in a macroscopic way, without getting to the gate level as was done with the PE.

3.5.1 CU General Structure

Figure 16 presents a diagram of the control unit structure. The components are:

a) CU Memory (CUM), which is a conventional, high speed random access memory in which SPEAC's instructions and CU data are stored. It can be replenished from mass memory and is accessed by the central processing unit and by the instruction lookahead unit.

b) Instruction Lookahead Unit (ILA) which fetches instructions from CUM and sends them to the instruction decoding unit. Since CUM is very fast, a sophisticated ILA is probably not necessary.

c) Instruction Decoding Unit (IDU) which performs basic instruction decoding and central indexing. The instructions are identified as CU, PE, or I/O instructions and sent to the respective instruction processor along with their indexed addresses and other data.

d) Central Processing Unit (CPU) which is the CU instruction processor and responsible for the execution of CU instructions. It

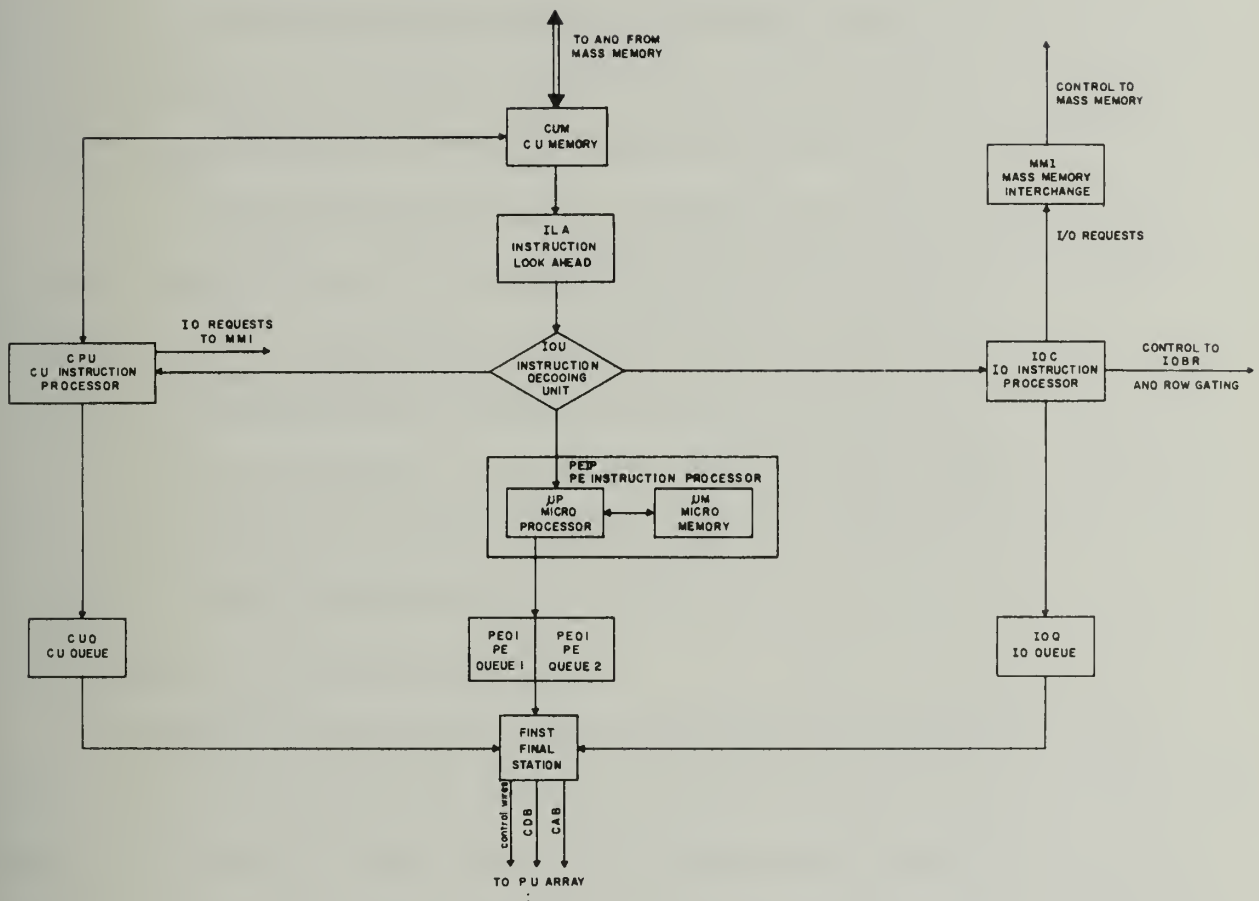


Figure 16. CU Structure

is basically a fast, highly parallel unit similar to one of ILLIAC IV's PE's. It should be compatible with the data formats used in the PE's. Therefore, for maximum versatility, it should also be microprogrammable like the instruction processor. The CPU is not completely independent from the PE array since it can send common operands to all PE's via CDB ("broadcasting") and also can receive data from the PE's. For this purpose, the CPU can send microsequences to the PE's via the CU Queue.

e) I/O Instruction Channel (IOC) which is the I/O instruction processor and executes array I/O instructions. Like the other two instruction processors, it could be microprogrammable for maximum versatility. The IOC sends I/O requests to the mass memory interchange and control pulses to the row gating and I/O Buffer Register (IOBR). It can also send microsequences to the PE via the IO Queue.

f) PE Instruction Processor (PEIP) which is the third and last instruction processor, in charge of PE instructions. It is fully microprogrammable and can be divided into two parts. The first part is a microprocessor (μP) which executes the microprograms and sends microsequences to the PU via two queues--PE Queue 1 and PE Queue 2. The second part is a micromemory (μM) which stores the microprograms. μM does not have to be a separate memory; part of CUM may be used as micromemory if this is the most economical scheme.

g) Four Queues which are: Queue (Q), PE Queue 1 (PEQ1), PE Queue 2 (PEQ2), and IO Queue (IOQ). These queues store microsequences

sent by each instruction processor, absorbing fluctuations in the rate of generation of these microsequences which enables the final station to keep the array as busy as possible.

h) Final Station (FINST) which analyzes the entries at the bottom of each queue and decides which microsequences to send to the array for optimum PE performance. It must also combine two queue entries into one PE microsequence since each queue entry is not a complete μsequence but a request to use one of the two pairs of buses in the PE's. FINST action will be explained in considerable detail in Section 3.5.3.

i) Mass Memory Interchange (MMI) which utilizes the several modules of mass memory in an optimum fashion, solving memory request conflicts. It receives requests from the following sources: CUP, IOC, Corner Memory and Peripherals.

3.5.2 Machine Synchronization - Events

Events are the means of synchronization in the machine; not only are they accessible to the user for problem-dependent synchronization (I/O and operations, for example) but they are also used by the microprograms to synchronize different microsteps executed in the PE's, CU and IOC. Each event is assigned an absolute number and it is basically a flip-flop; when OFF, the event did not occur and when ON, the event has occurred. A reasonable number of events are needed; 64 as a first approach, for example.

Therefore, synchronization is obtained with commands to "WAIT on event N" or "CAUSE event N." WAIT and CAUSE commands are attached to instructions and are recognized and obeyed at three units: CUP, IOC and FINST. Consider, for example, a CU instruction which needs as one operand a PE value sent via

CDB. The instruction goes to CUP which does any local processing needed and then issues the microsequence to CUQ. The microsequence contains a "CAUSE event N." The CU then idles on a "WAIT on event N." When the microsequence is executed; i.e., when the data needed from the array reaches the CUP, event N happens and CUP finishes execution of the instruction. This waiting time could be used by the CU for multiprocessing a serial program (a compilation, for example) being run simultaneously. One must make sure that an event will not be considered "occurred" because the FF is ON from another use of the same event number. Therefore, the user does have the responsibility of "releasing" an event when the present use of that event number terminates. This may be done when the event is waited on for the last time, with a special type of wait--WAIT and RELEASE--or an event may be specifically reset with a RESET EVENT command.

The following event manipulation commands are desirable:

- Wait on a boolean function of events
- Cause an event depending on a boolean function of others
- Cause several events simultaneously.

Basically those commands are for program use only since microsequence synchronization must be very fast and must be done with single events.

It should be noticed that one would never wait on a boolean combination of events since this would require the boolean function to be evaluated at each clock to determine if the wait is over. The way to do this is to have after each cause of the events that appear in the boolean function, a statement that evaluates the boolean combination and places the result on an extra number: N. Then the wait is simply on event N.

Care must be taken to avoid re-use of an event before its previous

use is completed. Certain complicated cases may be confusing. Consider, for example, the following program:

```
Input 1          cause event #3 ─┐
:                                │
:                                │
PE-multiply      wait on event #3 and release it
:
:
Input 2          cause event #3 ─┐
:                                │
:                                │
CU operation     wait on event #3 ─┘
```

In the situation above, Input 1 may occur and cause event #3. Then, before the PE-multiply or Input 2 occur, the CU operation may be executed and event #3 is ON so there is no wait.

The possibility of symbolic event names handled by the hardware could be investigated; the hardware would automatically assign symbolic event numbers to the first available physical event flip-flop. This would free the user of keeping track of which events are available and also no set of events would have to be reserved for μsequence use. However, the user would still have to release events.

Note also that with the present scheme, it is necessary to divide the events into two sets: user events and internal events. The latter will be used by the microprograms to synchronize the execution of microsequences.

3.5.3 Queue System and FINST

Queue entries can be considered as requests to use part of a PE. These requests are serviced by FINST which, if possible, combines two entries from different queues into a PE microsequence and sends the microsequence to the PE's. The purpose of FINST and the queue system is to keep both pairs of PE buses (A1, D1 and A2, D2) as busy as possible.

The basic principle involved is dynamic bus allocation; i.e., each queue entry does not ask specifically for use of bus 1 or 2, it asks for either a) any bus, or b) the bus that has access to the PEM module containing the address stored in X_i (i=1,2, or 3). Requests of type a are made for inter-register transfers, in which it is immaterial which bus is actually used; requests of type b are necessary for memory transactions since for these a specific bus must be used. Therefore, under dynamic bus allocation, CUP, PEIP and IOC do not specify the microsequences completely--FINST will dynamically allocate buses to the partial microsequences in the best possible way.

3.5.3.1 Queue Structure

Each queue entry contains basically a partial microsequence and information which is used by FINST. The fields of a queue entry are illustrated in the upper part of Figure 17. All four queues have the same structure although only Queue 2 has been detailed.

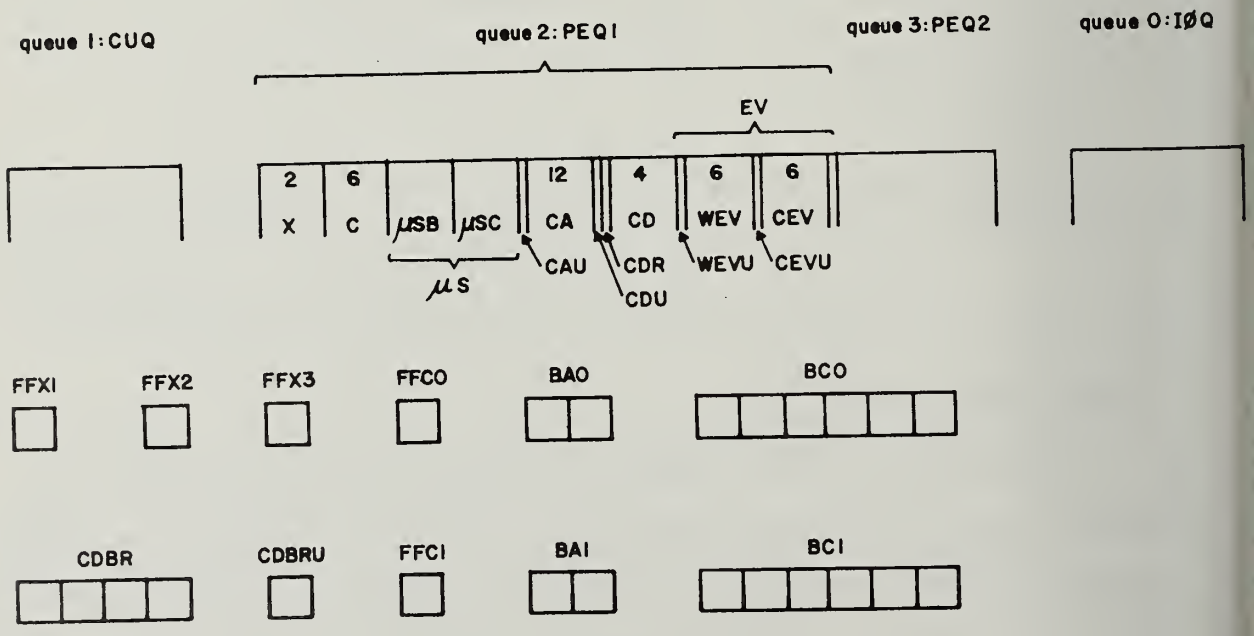


Figure 17. Queues and FINST Structure

The fields are as follows:

X: address field (2 bits). 0 means the address register is not used; i.e., we have a data transfer and not a memory fetch. X=i (where $1 \leq i \leq 3$) means the address register X_i in the PE's will be used in this microsequence.

C: counter field (~6 bits). 0 means the microsequence is a no-op. C=i>0 means that when a bus is assigned to that queue, then this microsequence and the next n-1 will be processed consecutively.

μS: these are fields that contain the partial-microsequence.

μSB: bus-dependent microsequence field (~23 bits). This is the part of the microsequence related to bus used.

μSC: bus-independent microsequence field (~55 bits). This is the part of the microsequence related to control that does not use buses.

CAU: use of CAB field (1 bit). CAU ON means CAB will be used and must be set to the value stored in CA.

CA: common address field (12 bits). This contains the value to be used as common address.

CDU: use of CDB field (1 bit). CDI ON means CDB_{in} will be used and must be set to the value stored in CD.

CDR: common data receive field (1 bit). When ON, CDB_{out} will be used to receive data from the PU's; this data must be stored in CDBR.

CD: common data field (4 bits). This contains the value to be used as common data.

EV: these are fields that control events.

WEVU: wait event use field (1 bit). When ON= this entry must await an event whose number is stored in WEV.

WEV: wait event field (~6 bits). This contains the number of an event

to be waited on.

CEVU: cause event use field (1 bit). When ON, this entry must cause an event whose number is stored in CEV.

CEV: cause event field (~6 bits). This contains the number of an event to be caused.

The bus-dependent microsequence field must be further explained. It can be divided into two sub-fields: μSBa and μSBb. μSBa, with 8 bits, corresponds to the control wires to gate into buses D and A (3 wires for each) and to control PEM (2 wires). In the actual microsequence, this field appears twice: once for each bus pair. μSBb, with about 15 bits, corresponds to the control wires to gate from buses D and A. The values of the bits in this field of a queue entry have a special meaning: a ZERO means that the corresponding control is not used in this microsequence and a ONE means that the control is used (i.e., the final microsequence must have in that position the appropriate bit to load from the bus that has been assigned to that queue entry.

3.5.3.2 FINST Structure and Operation

The structure of the final station will not be presented in detail; only the major registers and their uses are discussed and a few considerations are offered on the output logic of FINST (i.e., the part that merges together two queue entries and assembles the microsequences).

The major registers of FINST are illustrated in Figure 17 and are as follows:

FFXi (i=1,2,3): address control FF (1 bit). FFXi = j means that in the array, all Xi registers have addresses pointing into memory

module j (j=0,1). These flip-flops are automatically set by the CU (i.e., the FINST) every time a microsequence is sent in which the bit that controls gating into Xi is ON. The setting is based on the contents of the CA field in that microsequence. Local modifications (as in local indexing) of Xi cannot change the module it points to. This condition can easily be checked within each PE and causes an interrupt (just monitor the carry from the address registers). Besides the automatic setting, FFXi should also be settable by the programmer for special applications.

FFCi (i=0,1): conflict FF (1 bit). These are the conflict flip-flops, set either when the bus could not be assigned or when one or two of the bus assignments is not used on a particular clock because of bus conflicts or because the queue is empty.

BAi (i=0,1): bus assignment register (2 bits). When BAi = j, bus i is assigned to queue j. $j \in \{0,1,2,3\}$.

BCi (i=0,1). bus counter (~6 bits). When BCi = j, there are j microsequences left to be performed before the bus can be reassigned; BCi = 0 means that bus i is idle.

CDBR: common data bus register (4 bits). This is the register where values placed in CDB_{out} by the PE's are stored.

CDBRU: common data bus register use (1 bit). When equal to 1 it means that CDBR is in use; i.e., a result placed in it has not been removed by the CU and therefore CDBR cannot be reused before the CU frees it by resetting CDBRU.

The FINST decision procedure is now described: at each clock, FINST must decide to which of the four candidates the use of the PE buses will be

assigned. Once a request from a queue is granted, the next (C) requests from that same queue must be obeyed before the bus can be reassigned (where (C) is the contents of the counter field). This ensures the microprogrammer that, once control is obtained, it will be retained for a number of microsequences enabling the completion of a procedure before a new bus assignment destroys needed data. Therefore, groups of microsequences that must be executed sequentially, without interruption, are "linked" together by placing in the counter field of the first queue entry the number of microsequences in the group.

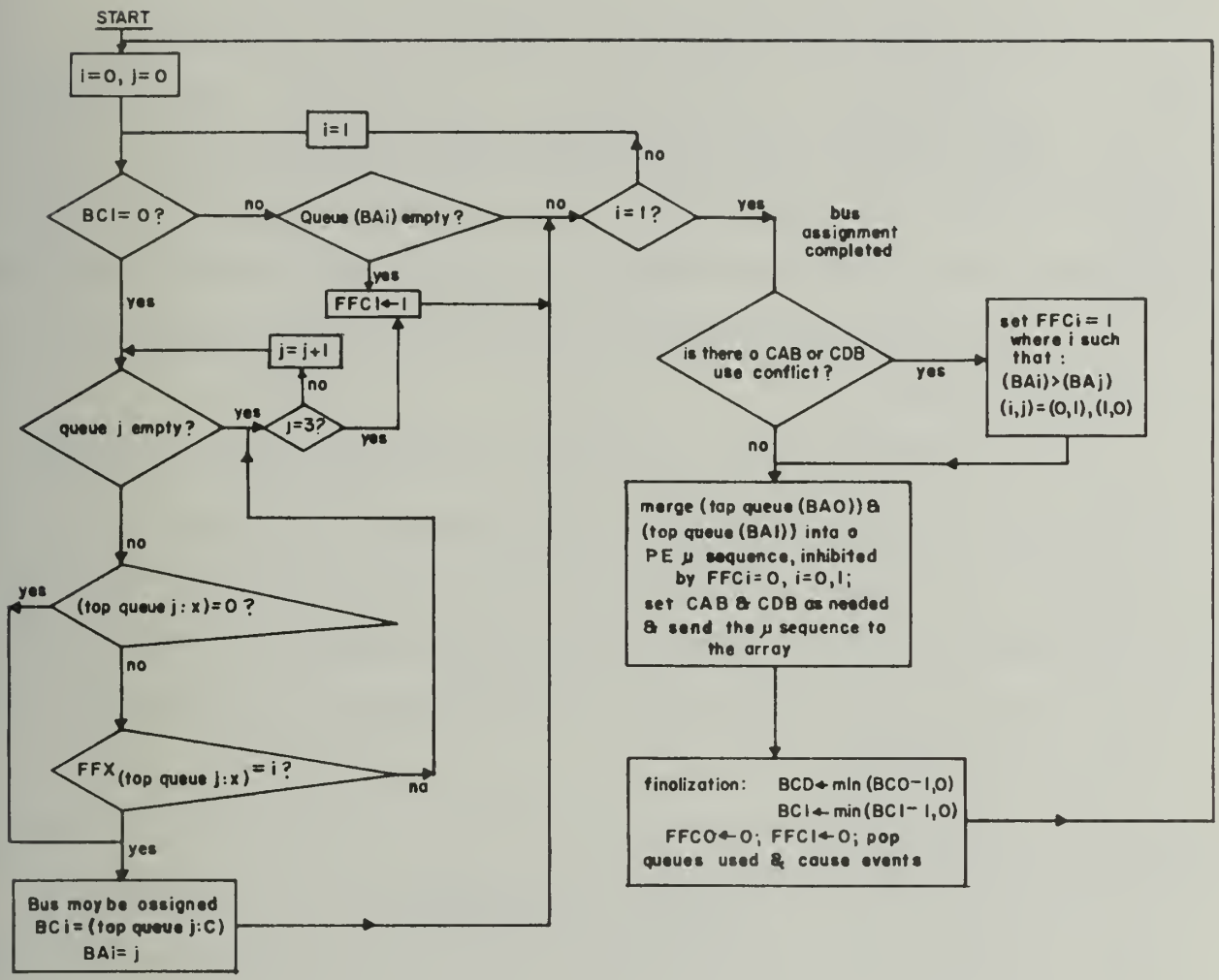
The FINST decision procedure is illustrated in Figure 18 by a flowgraph. If a bus counter register in FINST is zero, the corresponding bus is idle and an attempt is made to assign it. The order in which assignment attempts are made is, in Figure 18: IOQ, CUQ, PEQ1, and PEQ2. This attempts first to get the I/O done. This assignment hierarchy, in an actual implementation, would probably be dynamic and selectable by the programmer instead of fixed. Section 3.5.5 discusses a situation in which a dynamic assignment hierarchy is required.

The following observations should be made with respect to the flowgraph in Figure 18:

- The notation (Top Queue j:C) means the contents of field C of the entry at the top of Queue j.
- A queue is empty either when it is physically empty or when it is flagged WAIT on an Event that has not occurred yet.
- There is a CAB or CDB conflict when the following expression (where TQi means top queue i) is true:

a) (TQ(BA0):CAU)=1 AND (TQ(BA1):CAU)=1 AND (TQ(BA0):CA)≠(TQ(BA1):CA)

OR



BUS i i = 0,1
QUEUE j j = 0,1,2,3

Figure 18. FINST Action Flow-graph

(b) (TQ(BAO):CDU)=1 AND (TQ(BA1):CDU)=1 AND (TQ(BAO):CD)≠(TQ(BA1):CD)

OR

(c) (TQ(BAO):CDR)=1 AND (TQ(BA1):CDR)=1

OR

(d) ((TQ(BAO):CDR)=1 OR (TQ(BA1):CDR)=1) AND CDBRU=1

where the term (a) takes care of CAB conflicts, the term (b) detects CDB_{in} conflicts, the term (c) detects CDB_{out} conflicts, and the term (d) takes care of CDBR use conflict (i.e., CDBR has not yet been used after being set by a previous operation).

It should be pointed out that the decision procedure outlined in Figure 18 is only a basic algorithm. A few sophistications would have to be introduced in an actual implementation; specifically: a) the procedure should also be able to handle efficiently microsequences that do not require the use of any bus, and b) the possibility of deadlock should be considered and steps taken to avoid it.

Figure 19 illustrates the part of FINST that merges the two selected queue entries together and "assembles" the microsequence. Gate control selects which of the four possible inputs to each bus is actually gated into the bus; queue i is gated into the bus if i is the value of the expression written in each gate control box. Briefly, the assembly procedure is as follows: CDB_{out} is gated into CDBR if the CDR field of any of the two selected queue entries is ON; CDB_{in} is set from the CD field of the selected entry, if any, that has field CDU ON; CAB is obtained from the CA field of the selected entry, if any, that has field CAU ON. Field μSC of the final microsequence is the OR of these fields in the two selected entries. A check for conflicts would be necessary at this point to make sure that the two μSC fields are

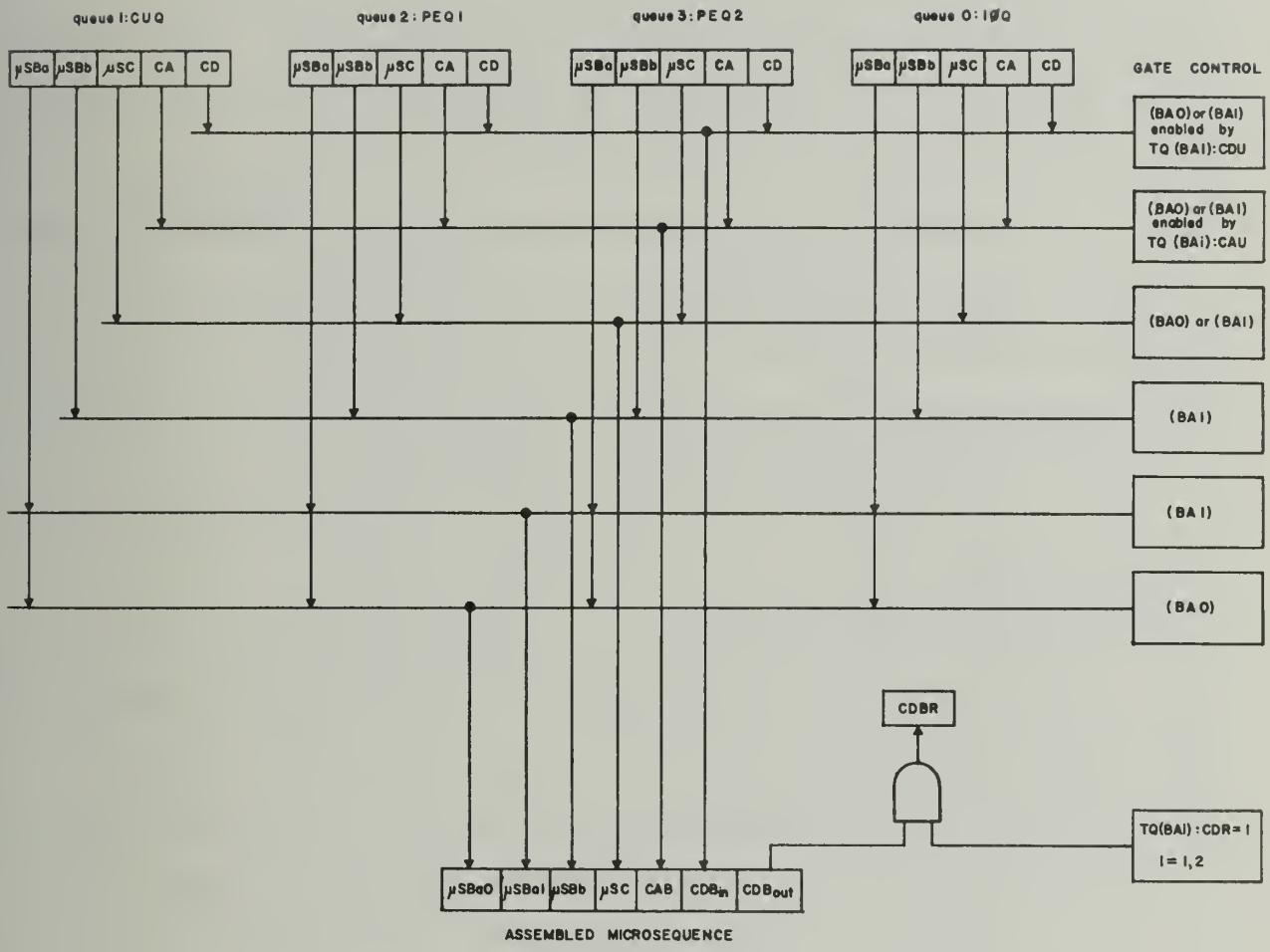


Figure 19. Final Microsequence Assembly in FINST

compatible to be OR'ed together; i.e., the actions determined by one of the entries must not conflict with the actions determined by the other. As explained previously, field μSBa appears twice in the microsequence, once for each bus pair. Therefore, μSBa0 is obtained from the μSBa field of the entry selected by BA0 and μSBa1 is obtained from the μSBa field of the entry selected by BA1. Finally, field μSBb is simply taken out of field μSBb of the entry selected by BA1. A conflict is also possible at this point: fields μSBb of the two selected entries should yield a zero when AND'ed together, bit by bit. If this is not the case, there is a conflict in the YSBb fields. It should also be pointed out that every gate control box is inhibited by the conflict flip-flops FFCi; i.e., when FFCi is ON, no field from the entry selected by BAi is used in the assembly of the microsequence.

3.5.4 The PE Instruction Processor

The basic structure of the PE instruction processor is presented in Figure 20. The components are:

a) A macro-instruction register (MIR) which holds the op code and variant field of the macroinstruction being processed. This register is initialized by IDU and is accessible to the microprocessor to be used in controlling microprogram fetch and in arithmetic and masking operations.

b) A microinstruction register (μIR) which holds the op code and addresses of the microinstruction being executed.

c) A micro-memory (μM) which holds the microprograms.

d) A PEIP busy flip-flop (PEIPB) which is turned ON by IDU when a macroinstruction is delivered to the microprocessor and is turned

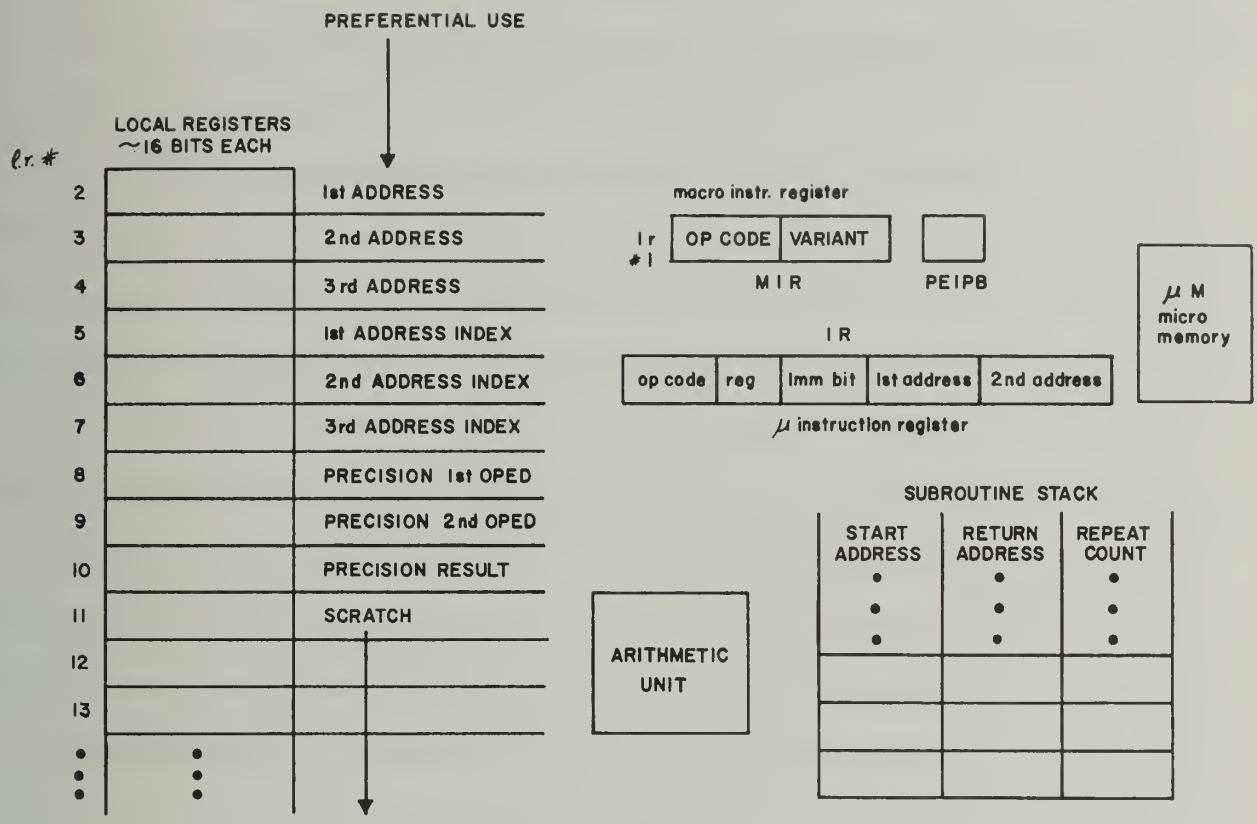


Figure 20. Basic PEIP Structure

OFF by the PEIP logic when the last microinstruction of the macroinstruction has been processed. This signals IDU that the microprocessor is idle and ready to receive the next macroinstruction.

e) A subroutine push-down stack used in controlling execution of subroutines by the microprocessor. Each entry in the stack contains three fields: a start address field which holds the address in which the subroutine starts; a return address field which holds the address of the first instruction following the subroutine; and a repeat count field containing the number of times the subroutine is to be executed.

f) A group of local registers which is used to hold intermediate results in arithmetic operations. The contents of the local registers can be used in assembling the different fields of the partial microsequences to be fed into the PE queues: PEQ1 and PEQ2. Finally, the local registers are also accessible to the IDU which initializes them with the instruction addresses and other instruction data. In this connection, MIR can be considered a local register and it is assigned local register number 1. The other local registers are numbered in sequence and they are accessed by their local register number. Sixteen local registers are proposed, each 12 to 16 bits long.

g) An arithmetic unit capable of performing fixed-point operations on short words: 12 to 16 bits is enough. At least addition, subtraction and multiplication are available (integer division and module operations are also useful). The operands are

either the contents of specified local registers or literals.

The results are placed in a specified local register.

An arithmetic unit is needed to enable microprograms to accept dynamically specified parameters as word length, number of addresses, etc., since it is obviously extremely inefficient to have one complete microsequence stored for each small variant of a basic instruction.

This also determines the need for a number of relatively sophisticated microinstructions; for example, subroutine calls. The suggested microinstruction repertoire is presented in Table 8. This repertoire allows very efficient microprograms with respect to μM use. It is assumed that the microprocessor is fast enough to allow an average output of one partial microsequence each 100 nsec. Fluctuations in this rate are absorbed by the queues.

As indicated in Figure 20, the microinstructions' format uses four fields: op-code, local register number (LR), immediate bit (IMM), and two addresses, A1 and A2, each as long as a local register. The use of these fields for each microinstruction is detailed in Table 8. The immediate bit qualifies the first address; if IMM is ONE the first address contains an immediate operand instead of a local register number.

The partial microsequences are generated in pairs, assuming optimal conditions; i.e., assuming that both buses will be available. The first partial microsequence in each pair is placed in PEQ1 and the other one is placed in PEQ2 so that if both buses are available they will be executed simultaneously and if not they will be executed sequentially. Events are used to coordinate the draining of the queues as needed. One extra bit in the queues may be needed to signal a request for the simultaneous execution of a partial microsequence from PEQ1 and one from PEQ2 as is required in a swap of registers.

Op Code (mnemonic)	Description
CALL	Subroutine call; executes (A2) times the subroutine starting at μM address (A1)
RETURN	Marks the end of a subroutine or the end of a microprogram.
GOTO	Transfers control to the microinstruction in μM address (A2).
IF	If (LR) masked by (A1) is all 1's then transfers control to the microinstruction in μM address A2
ADD	Add (A1) and (A2) and place the result in LR
SUB	Subtract (A1) from (A2) and place the result in LR
MULT	Multiply (A1) and (A2) and place the result in LR
μSEQ	Emit a partial microsequence to PEQ1 or PEQ2

Table 8. Microinstruction Repertoire

This will also necessitate a change in assignment hierarchy or else the array will idle for a long period waiting for both buses to become available.

The microinstruction μSEQ must be able to "assemble" a partial microsequence (placing in each field either a literal or the contents of a specified local register) and place it either in PEQ1 or PEQ2.

Therefore, this microinstruction is unreasonably large and requires about 100 bits of data. This shows the need for a microinstruction with a variable number of bits (just as is the case of macroinstructions) to optimize memory use since the μSEQ microinstruction takes so much more space than the other microinstructions.

3.5.5 IDU and Instruction Format

Central indexing is decoded and performed by the IDU which hands the resulting addresses to the three instruction processors. The detailed instruc-

tion format is illustrated in Figure 21. Instructions are composed of a variable number of "chunks," each 12 to 16 bits long. A chunk may be an address, an op code or some other type of data. The smallest instruction contains only two chunks: IDU information and op code.

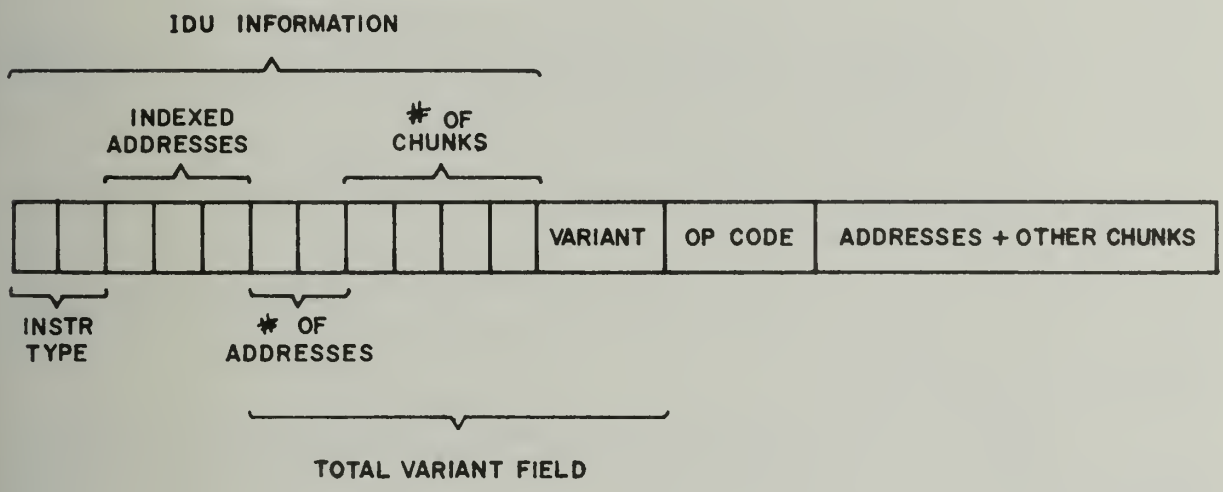


Figure 21. Detailed Instruction Format

The four fields in the first chunk (11 bits) contain information used by IDU:

a) The instruction type field, with 2 bits, indicates whether the instruction is a CU, IO or PE instruction enabling IDU to send the instruction to the appropriate processor.

b) The indexed addresses field has 3 bits. If bit i is on, then

the i^{th} address is to be indexed. The following convention is adopted for the order in which base addresses and index addresses are presented:

third chunk: first base address

fourth chunk: if first address is indexed, then it is the address index for the first address, else it is the second base address.

:

etc.

c) The number of addresses field indicates how many of the chunks following the first two are addresses.

d) The number of chunks field gives the total length of the instruction.

These last two fields are also sent as part of the variant field since they are needed by the processors.

IDU places an instruction in an instruction processor as follows: initialize instruction register with op code and total variant field; initialize the three first local registers with the addresses, but do not change a register to which an address was not given in the present instruction; then initialize the next local registers with the extra chunks in the order given--the instruction processor decides what to do with them.

3.6 Mass Memory

A survey was conducted on the state of the art of mass storage systems including bulk magnetic core, fast disks, fast drums and semiconductor memories. Fast magnetic drum (at one-half cent per bit) or disk (as low as one-twentieth of one cent per bit) could be used as the mass memory since they

have a significant price advantage over the other two systems. However, being cyclic, these systems would introduce synchronization problems and/or latency time waits. Therefore, while disks are still being considered as a possible very-large-capacity back-up for mass memory, the choice for the actual mass memory is a random access system: bulk core or semiconductor.

CDC bulk core model 6636 was picked up as a sample of what is now available. Its characteristics are:

- 7.5 million bits per module
- the maximum number of modules is four
- cycle time: 3.2 μsec; access time: 1.6 μsec
- up to four modules can be interleaved
- the transfer rate is 25 to 100 million 6-bit chars per second
- it fetches in long words of 480 bits
- its cost is approximately three cents per bit.

It is expected that in the near future, price of bulk core will drop to below one cent per bit. Assuming the availability of units of this price and with cycle times as above, a unit fetching in 512-bit words could be used as SPEAC's mass memory.

As for semiconductor memories, the main advantage core has over any semiconductor type is the ability to be non-volatile. Semiconductor memories are already available for less than three cents per bit although the price always goes up for special configurations like the long word that is needed in SPEAC's mass memory. Since semiconductor is so much faster than bulk core, one might attempt to multiplex a narrower word but faster semiconductor memory to achieve the desired word length and access time. In addition, a large memory of shift registers might be considered. A special design would be easier to

achieve and control can be maintained over synchronization and latency problems.

Therefore, mass memory will be a random-access unit: bulk core or semiconductor, depending on economic considerations. It is assumed that several modules of mass memory will be overlapped under the control of the mass memory interchange (MMI) so that conflicts between mass memory access requests from different sources will be infrequent. An average cycle time of 2 μsec (1 μsec access time) for the mass memory has been assumed in all timing estimates.

3.7 I/O Buffer Register

The structure of the I/O buffer register (IOBR) is illustrated in Figure 22.

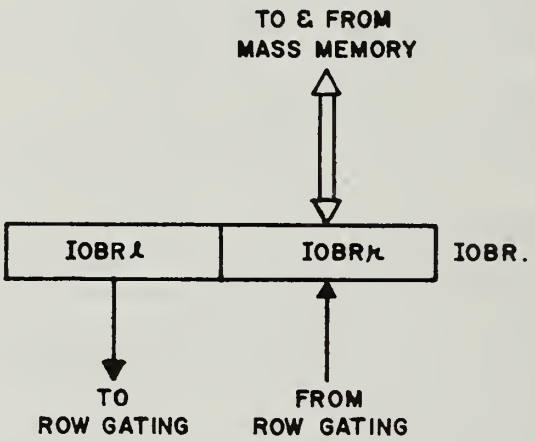


Figure 22. I/O Buffer Register Structure

The register is divided in two parts: a right part (IOBRr) and a left part (IOBRℓ). Each part is as long as a mass memory word: 512 bits. IOBRr is connected to the mass memory and is the actual buffer register; it can also receive data from the row gating (128 hexadecimal digits, one from each PE in a PE row). IOBRℓ is needed to achieve routing capability in SPEAC; it

can send data to the row gating. IOBR as a whole can be shifted end around, left or right in 4-bit (one hexadecimal digit) increments. In order to achieve good routing speed, it is vital that IOBR can be shifted by any distance (from 1 to 127 digits) in only a few clock periods. This poses an interesting minimization problem: how many direct shift paths should be implemented in order to obtain any shift in a given number of clocks? Also, a few distances are especially important and the corresponding shifts should be particularly fast; this is the case with powers of two since routes by a power of two appear much more frequently than other routing distances as they are used in log-sums, Fast Fourier transforms, etc. Finally, there is the important economic restriction of keeping the number of direct shift paths at a minimum since for each path one needs roughly one gate per bit and there are 1024 bits in IOBR. It was decided that a minimum of 7 direct shift paths are needed with the following direct shift distances: 128 left (this is vital to the operation of both I/O's and routes), 1 right and left, 32 right and left, and 8 (or 4) right and left. This scheme enables one to perform any shift in not more than 7 clocks. The worst case is distance 52 (50 if one uses 4 instead of 8). Moreover, shifts by a power of two take not more than 4 clocks and most take only one or two. At a cost of 2K more gates, one could implement 9 direct paths (128 left, 1 left, 1 right, 2 left, 2 right, 8 left, 8 right, 32 left, and 32 right) for a worst case shift of 5 clocks.

It is assumed for the remainder of the paper that 7 paths were implemented. This represents an investment of about 12K gates in IOBR which is a reasonable price to pay to achieve routing and I/O buffering for the whole machine. Table 9 presents the number of elementary shifts needed to shift a number by any distance from 1 to 64 when the direct paths are: 128 left,

A*	B*	A	B	A	B	A	B	A	B	A	B	A	B	A	B
1	1	9	3	17	5	25	4	33	2	41	4	49	6	57	5
2	2	10	4	18	6	26	4	34	3	42	5	50	7	58	5
3	2	11	4	19	5	27	3	35	3	43	5	51	6	59	4
4	1	12	3	20	4	28	2	36	2	44	4	52	5	60	3
5	2	13	4	21	5	29	3	37	3	45	5	53	6	61	4
6	3	14	5	22	5	30	3	38	4	46	6	54	6	62	4
7	3	15	5	23	4	31	2	39	4	47	6	55	5	63	3
8	2	16	4	24	3	32	1	40	3	48	5	56	4	64	2

*A - shift distance
*B - number of elementary shifts

Table 9. Number of Elementary Shifts for Each Shifting Distance

1 left, 1 right, 32 left, 32 right, 4 left, and 4 right.

4. SPEAC's OPERATION

4.1 Generalities - Data Format

The algorithms used in performing the most important instructions will be outlined in this section and timing estimates will be presented. The timing is based only on a count of the PE clocks necessary to perform the instruction; no CU delays were taken into account. Therefore, the estimates neglect CU instruction fetching, decoding and central indexing times. Also neglected is the time taken by the CU to execute microprogram control instructions; i.e., microinstructions that do not generate microsequences. These approximations are justified by the assumption that CU is, on the average, faster than the PE's (CU clock rate is about twice PE clock rate) and the queues insure that PE's will not have to wait by CU.

The timings are also a function of how much overlap is possible when the instruction is executed; i.e., how many buses are available for the PE instruction use. This factor depends on the assignment hierarchy used by FINST, on the location of the operands in PEM and on how much I/O is taking place when the instruction is executed. In the timings, at least one bus is assumed always available for PE instructions (or else the worst case times will obviously be infinity). Sometimes two timings are given: the "normal" one, with only one bus available and the "optimum" timing, assuming maximum overlap (two buses are available). CDB and CAB bus conflict is also a possible cause of delays which were not taken into account in the times since they depend on how much I/O is going on. However, these delays are expected to be negligible in a PE with three address registers.

As discussed in Section 3.1 - g, the machine accepts any word format,

since there is nothing in the hardware to "freeze" the data representation. Of course, adequate microprograms must be written to deal with a desired word format.

An arbitrary (and quite conventional) format for floating-point numbers was picked up and used in the timings. This representation will be called the "standard format" and is as follows: a number appears in PEM as indicated in Figure 23.

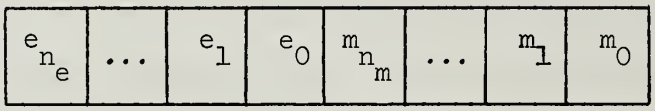


Figure 23. Standard Floating-Point Format

Each PEM location contains one hexadecimal digit. The location of m_0 is low memory address. There are $N_e = n_e + 1$ exponent digits and $N_m = n_m + 1$ mantissa digits. Mantissa is in sign and magnitude and the sign is in bit e_{00}; i.e., the low order bit of the LSD of the exponent. Therefore the exponent has $4N_e - 1$ bits since one bit of the exponent is used for mantissa sign. The exponent base is 16 and the exponent is represented in excess notation. The number A represented in Figure 23 has a value given by:

$$A = (-1)^{e_{00}} \times (m_{n_m} (2^{-4}) + \ldots + m_1 (2^{(-4)n_m}) + m_0 (2^{(-4)(n_m+1)}))(16^E)$$

where E, the exponent, is given by:

$$E = e_{01} + (e_{02})(2) + (e_{03})(2^2) + (e_1)(2^3) + \ldots + (e_{n_e}) (2^{4n_e-1}).$$

If a floating-point number is normalized, $m_0 \neq 0$; i.e., at least one of the four bits of m_0 is one.

A particularly important length for a floating-point number is 32 bits, which was often taken as the standard floating-point number in this

section. A 32-bit floating-point number has one mantissa sign bit, a base 16 exponent with 7 bits and a 24-bit mantissa.

4.2 Local Indexing

Operand addresses are sent to one of the PE address registers via CAB. Only one clock is needed to transmit an address in this fashion. Then, if required, any address may be locally indexed at a maximum cost of 1.6 μsec (16 PE clocks) per indexing.

The microsequence to perform local indexing is presented in Table 10; the notation is explained in the introduction of Appendix B. It is assumed that the address to be indexed ($x_2x_1x_0$) is loaded in X_1 and the index is $i_2i_1i_0$.

In conclusion, local indexing is relatively fast (about 7% of the time for a 32-bit floating-point multiplication) and the procedure does not penalize the users that do not need it since it is performed only when the instruction variant field is adequately set. Also, the microsequence presented can be significantly speeded up if one knows that the index is less than three hexadecimal digits long.

4.3 Multiplication

Two mantissas A and B, each with N hexadecimal digits, are to be multiplied. Using the notation of expressions 1 and 2 in Section 3.2, the following steps are performed:

1) load a_0 from memory into register B
2) load b_0 from memory into register A_r
3) set to zero the remainder of register A; i.e., A_c and A_m
4) multiply a_0 and b_0 using four "add and shift" commands. At the

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
1	1	$X_2 \leftarrow$ CAB (address (i_o))	Put in X_2 the address of the index
2	1	$A_c \leftarrow X_1$	Transfer address to be indexed to A_c
3	2	B $\leftarrow$ PEM (X_2); shift A right 4	Fetch i_o and place x_o in A_m
4	3	Wait for PEM fetch	
5	4	Wait for PEM fetch	
6	6	$A_r \leftarrow (B+A_m)$; C_n=0; lcFF4 $\leftarrow C_{n+4}$; Incr X_2	Add i_o and x_o and place in A_r
7	6	B $\leftarrow$ PEM (X_2); shift A right 4	Fetch i_1 and place x_1 in A_m
8	7	Wait for PEM fetch	
9	8	Wait for PEM fetch	
10	9	$A_r \leftarrow (B+A_m)$; C_n=lcFF4; lcFF4 $\leftarrow C_{n+4}$; Incr X_2	Add i_1 and x_1 and place in A_r
11	10	B $\leftarrow$ PEM (X_2); shift A right 4	Fetch i_2 and place x_2 in A_m
12	11	Wait for PEM fetch	
13	12	Wait for PEM fetch	
14	13	$A_r \leftarrow (B+A_m)$; C_n=lcFF4; lcFF4 $\leftarrow C_{n+4}$	Add i_2 and x_2; shifting A will place x+i in A_c from which it is returned to X_1; an overflow in the indexing causes an interrupt.
15	14	Shift A right 4; interrupt on lcFF4 ON	
16	15	$X_1 \leftarrow A_c$	

Table 10. Microsequence for Local Indexing

end of this step, A_m and A_r will contain the two-digit product of a_0 and b_0; b_0 was destroyed and A_r now contains m_0

5) if a double precision product is desired, store $m_0 = (A_r)$ into memory; jump this step if a single precision product is to be obtained

6) increment by 1 the contents of register X_2 (it is assumed that initially X_1 contains the address of a_0 and X_2 contains the address of b_0). Therefore, X_2 now contains the address of b_1

7) load b_1 from memory into reg A_r

8) multiply b_1 (in reg A_r) and a_0 (in reg B) as described in step 4; note that the "carry" of the previous multiplication is automatically added to the product

9) increment X_1 by 1, decrement X_2 by 1

10) shift register A left 4 bits which vacates A_r

11) reload register A_r and B; multiply

12) A_r now contains m_1 which can be stored or discarded

⋮

And so on, following the algorithm of Section 3.2. To determine digit m_i of the product ($i \leq n$), the cycle: [increment X_1, decrement X_2, load B, load A_r, multiply, shift left 4] is repeated i+1 times. On the first cycle only one increment-load is performed and on the last cycle there is no shift left 4. It has already been mentioned that in single precision, product m_n is the first digit of the product which is not discarded and it can be stored in b_0's position (or a_0's). If at the end of the multiplication m_{2n+1} does not equal zero, a normalization is needed; each hexadecimal digit is read and restored shifted right. Therefore, m_n is discarded, m_{n+1} becomes the low

order digit and m_{2n+1} the high order one.

This algorithm is general and can handle mantissas with any number N of digits. The introduction of the scratchpad memory, however, results in a remarkable improvement in the procedure, especially for N not greater than 16 (which includes most practical applications).

The method consists of overlapping a multiplication of two digits with the fetch of a third digit which is temporarily stored in sM and will be used in a subsequent multiplication. This is always possible because the "add and shift" command used in multiplication does not need any PE bus; a bus is thus left available for the fetch. Since the multiplication takes 4 clocks and a fetch only 3, there is still time to increment the address register used (preparing for the next fetch) and to reload B concurrently with its last use in an "add and shift." A fifth clock is required to reload A_r and a sixth if it is necessary to store A_r in sM before reloading A_r. The procedure described is listed in Appendix B, note a, under the name MF (for multiply and fetch). It should be also pointed out that the result is now first stored in sM and only after normalization is written in PEM which avoids the relatively slow process of rereading and restoring in PEM only to normalize.

The time required to multiply two mantissas (each N digits long) can now be estimated: N^2 executions of MF are required, taking 5 clocks each; N product digits must be stored, which takes 6 clocks per digit (see function ST in Appendix B, note d) and N more clocks to store the product temporarily in sM. Finally, about 13 clocks are necessary for initialization and control. Therefore:

$$T_m \cong 5N^2 + 7N + 13 \qquad , N \leq 16 \qquad (1)$$

where T_m is the time for mantissa multiplication in clocks. Since each clock

takes 100 nsec, for N = 8 a T_m of 40 μsec is obtained.

4.3.1 Floating-point Multiplication

The algorithm is relatively simple: initially the mantissas are multiplied as described previously and the normalized single precision product is stored. A_m is left with a 1 if normalization was performed (i.e., if $m_{2n+1} = 0$) and with a 0 otherwise. A_m is then subtracted from the first exponent and the second exponent is added to the difference which obtains the exponent of the result. Five extra clocks are needed to detect exponent overflow or underflow and to recode the exponent of the result in excess representation as explained in Appendix B, note f. The sign of the result is obtained from the exclusive - OR of the signs of the factors.

Timing estimate: for two floating point numbers with N_m digits in the mantissa and N_e digits in the exponent, the mantissa product will take (from (1)) about $5N_m^2 + 7N_m + 13$ clocks; exponent manipulation takes about 4 clocks per digit plus 6 clocks per digit for storage and about 5 clocks for control. The final expression is:

$$T_{fpm} \cong 5N_m^2 + 7N_m + 10N_e + 18 \quad , N_e + N_m \leq 16 \qquad (2)$$

where T_{fpm} is the time for floating-point multiplication in clocks.

For the "standard" 32-bit floating-point number, $N_e = 2$ and $N_m = 6$ which yields $T_{fpm} \cong 27$ μsec. For this case, the precise μsequence is presented in Appendix B and the results obtained are as follows: normal time = 25 μsec; optimum time = 24 μsec. Two 64-bit floating-point numbers ($N_m = 12$, $N_e = 4$) can be multiplied in about 86 μsec.

It should be remarked that the algorithm illustrated obtains the single precision product by truncation of the double precision product. If

simple truncation is not satisfactory and rounding is to be performed, then a small addition is needed in the microsequence. This is not too time consuming, however.

4.4 Addition and Subtraction

Unsigned addition or subtraction is quite straightforward and can be performed in the following steps:

1) load from PEM address (X_1) into register B.
2) load from PEM address (X_2) into register A_m.
3) add or subtract using input carry (C_n) zero (one in subtraction) for the first cycle and C_n=lcFF4 for the remaining cycles. Also, at each cycle, lcFF4 stores the output carry C_{n+4}. Therefore, at every cycle after the first one, lcFF4 contains C_{n+4} from the previous step.
4) increment X_1 and X_2 by 1.
5) go to step 1.

On the last cycle, lcFF4 is gated to the interrupt wire since lcFF4 ON (OFF in subtraction) at this point indicates an oveflow.

Timing estimate: for two unsigned fixed-point numbers with N digits each, one needs, per digit, 6 clocks to fetch the two operands, 1 clock to add and 5 clocks to write the result in PEM. Therefore:

$$T_a \cong 12N \qquad (3)$$

where T_a is the time for unsigned addition or subtraction in clocks. Thus, $T_a \cong 10$ μsec for N = 8 digits.

4.4.1 Signed Addition and Subtraction

There are several different ways to perform signed addition and

subtraction. Signed numbers can be stored in PEM either in a complement form or as sign and magnitude. The latter seems to be preferable since it speeds up multiplication and slows addition.

To add two signed numbers represented in sign and magnitude notation, it is necessary first to compare the signs. The result of the comparison is stored in lcFF1 which will be ON if the signs are equal, OFF otherwise. lcFF1 is then used to control whether an addition or a subtraction is actually performed. The two numbers are then added (or subtracted, if lcFF1 is OFF) and the final output carry, which is stored in lcFF4, is analyzed to determine the sign of the result, whether recomplementation is needed or not and if there was overflow. The rules are presented in Appendix C, note f.

Signed addition (or subtraction) takes 6 clocks per digit to fetch the two operands, one clock to add/subtract, one clock to temporarily store the result in sM (assuming $N \leq 16$, this is possible and speeds up recomplementation considerably), 2 clocks per digit to recomplement (PE's in which this operation is not needed are disabled), 6 clocks per digit to store the result in PEM and about 10 clocks for control and sign manipulations. Therefore,

$$T_{sa} \cong 16N + 10 \quad , N \leq 16 \qquad (4)$$

where T_{sa} is the time for signed addition or subtraction in clocks; for $N = 8$, $T_{sa} \cong 14$ μsec.

4.4.2 Floating-point Addition and Subtraction

The algorithm is quite complex and can be divided into six distinct phases: a) exponent comparison, b) exponent subtraction, c) hexadecimal point alignment, d) mantissa addition, e) recomplementation, and f) normalization.

The basic steps are the following:

1) Set up X_1 and X_2 with the addresses of the two exponents.

2) Fetch the exponents (storing them temporarily in scratchpad memory to avoid subsequent fetches) and compare them, exchanging the exponents and addresses in PE's with the "wrong" order so that all PE's will have in X_1 the address of the number with the larger exponent.

3) Compute the difference $\underline{d}$ of the exponents and add it to address X_2, thus performing hexadecimal point alignment.

4) Set up A_c with (FFF-N_m+1) via CAB and add $\underline{d}$ which prepares in A_c a trap that will overflow when N_m-d+1 is added to it; this will indicate that all valid digits of the smaller operand have been used and zeros must be substituted for the remaining digits.

5) Perform the actual addition following the algorithm described in the previous section with one extra step: after loading B from PEM, B is zeroed if a carry has already occurred in A_c. A_c contains initially the trap described in step 4 and is incremented by one as each pair of digits is added. lcFF2 is used to store the first carry from A_c. The sum is temporarily stored in sM for possible recomplementation and normalization before it is finally stored in PEM.

6) The final carry is analyzed to determine if there is a need for recomplementation or if an "overflow" occurred; i.e., if one extra MSD containing a ONE should be added to the mantissa. The rules are presented in Appendix C, note f.

7) Recomplementation is performed; only PE's in which this operation is necessary are enabled. The recomplemented result goes back

to sM.

8) X_2 and X_1 are used as counters: X_1 is initialized to FFF (all ones) and X_2 is initialized with the larger exponent. Then both registers are decremented by one for each leading zero in the mantissa of the result. Therefore, at the end of the process X_2 will contain the exponent of the result and X_1 will contain a trap to be used in A_c in the next step.

9) The mantissa of the result is written in PEM using X_1 to store the address of the result in PEM and X_2 to store the address of the result in sM. The mantissa is written from LSD to MSD and the trap in A_c is used to write initially as many trailing zeros as there were leading zeros before normalization.

10) The exponent is written from sM into PEM.

Timing estimate: since the procedure is so complex, it is quite difficult to obtain a precise formula for the number of clocks in addition. As a rought estimate, it takes for each pair of mantissa digits: 9 clocks to add, 2 clocks to recomplement, 2 clocks to count leading zeros and 8 clocks to write in PEM; for each pair of exponent digits: 3 clocks to compare, 3 clocks to subtract and 6 clocks to write in PEM. Adding about 50 clocks for control, sign manipulation and other housekeeping actions, the final expression is:

$$T_{fpa} \cong 21N_m + 12N_e + 50 \quad , N_e + N_m \leq 16 \qquad (5)$$

where T_{fpa} is the time for floating-point multiplication in clocks, N_m is the number of digits in the mantissa and N_e is the number of digits in the exponent. Thirty-two bit floating-point numbers with $N_m = 6$ and $N_e = 2$ take about 20 μsec to add. For this case, a <u>precise</u> microsequence is presented in Appendix

C and the results are: normal time = 21 μsec, optimum time = 19 μsec. Two 64-bit floating-point numbers can be added in about 35 μsec.

4.5 Other Operations

A few other important operations are now considered and a quick sketch is presented describing how they would be performed in SPEAC.

4.5.1 Division

This operation has not been considered in detail and while it is probably possible to design a sophisticated division algorithm that will use the PE very efficiently, this will take considerable research. On the other hand, even a very straightforward restoring division algorithm can be performed in an acceptable time. For $N_m \leq 8$, the two mantissas can be stored in sM; then the divisor is repeatedly subtracted from the dividend until a final borrow results and disables the PE. This is performed a maximum of 15 times; then all PE's add the divisor to the remainder to restore a positive remainder. The number of subtractions is counted in A_c. Each subtraction takes only 2 clocks per digit once the operands are in sM. Therefore, it takes at least $32N_m$ clocks to determine each digit of the quotient. Adding about 3 extra clocks per subtraction for control, one obtains the following rough timing estimate for mantissa division:

$$T_d \sim 32N_m^2 + 50N_m \qquad , N_m \leq 8 \qquad (6)$$

This yields about 130 μsec for 24-bit mantissa division and not more than 140 μsec for 32-bit floating-point division. The ratio of about six between floating-point division and floating-point multiplication times is adequate for this type of machine (in ILLIAC IV, this ratio is 7).

4.5.2 Logic Operations

Logic operations are quite straightforward in this machine since the A/L unit in the PE's can directly perform all sixteen logical functions of two variables. Therefore, to obtain any bit-by-bit logic function of two operands, each N digits long, the same algorithm described for unsigned addition (Section 4.4) can be performed; the timing is also as given by (3):

$$T_{\ell} \cong 12N \qquad (7)$$

where T_{ℓ} is the time required to perform one bit-by-bit logical operation.

4.5.3 Comparisons

In SPEAC, the result of a comparison is normally stored either on a lcFF or in the mode register. It can also be stored in sM or PEM for future use or sent to the CU via CDB. The six different types of comparisons ($>$, $<$, $\geq$, $\leq$, $=$, $\neq$) can readily be performed by the A/L unit. The algorithm for comparing two unsigned numbers is similar to the algorithm to add two unsigned numbers; as each pair of digits is compared, the result of the comparison for $=$ is always stored in lcFF1. This is needed even to perform a comparison for $>$, $<$, $\geq$, or $\leq$ since lcFF1 is used to "freeze" the result of the comparison once the first pair of unequal digits is found. For example, the typical microsequence for a $<$ compare is as follows:

- load first operand from PEM into A_m
- load second operand from PEM into B
- enabling on lcFF1 ON, store the comparison $A_m = B$ in lcFF1 and $A_m < B$ in lcFF4.

When all the digits have been compared, lcFF4 will have the resulting bit.

Therefore, the timing is:

$$T_c \cong 7N \tag{8}$$

where T_c is the time in clocks for comparisons of two unsigned numbers, each N digits long, leaving the result in the PE.

Signed and floating-point comparisons require a little more control but the linear dependence on N is as in (8). Rough estimates are:

$$T_{sc} \sim 7N + 10 \tag{9}$$

$$T_{fpc} \sim 7(N_E + N_M) + 20 \tag{10}$$

where T_{sc} is the time for signed comparisons in clocks and T_{fpc} is the time for floating-point comparisons in clocks.

4.5.4 Shifts

Shifts by a total distance of b bits are easily performed in two phases: address indexing is used to shift by (b div 4) and register A shifts are used to shift by (b mod 4). sM is also frequently used as temporary storage, especially in end-around shifts. If b is global (i.e., all PE's will shift by the same distance) then the address indexing is performed in the CU. In general, it takes in the worst case 3 clocks to shift each digit, one to store it in sM and 6 to store the shifted digit back in PEM. Therefore:

$$T_s \sim 12N \quad , N \leq 16 \tag{11}$$

where T_s is the time in clocks to shift a number with N digits by a global distance. The operation is a little more complex if b is local; i.e., the shifting distance is different in each PE. In this case, local indexing is initially performed, taking about 20 clocks, to "shift" by "b div 4." The quantity "b mod 4" is then stored in LC and three successive shifts are performed which are enabled by lcFF1, lcFF2 and lcFF3 respectively. The remainder of the operation is as for global shifts. Therefore:

$$T_{ls} \sim 12N + 20 \quad , N \leq 16 \qquad (12)$$

where T_{ls} is the time in clocks to shift a number with N digits by a local distance.

It should also be pointed out that the PE, besides shifting, has very good bit manipulation capability in general due to the locally controlled gating into A_m.

4.6 I/O

Both I/O and routing are performed using the row gating and IOBR. I/O will be described first. An elementary I/O operation consists of interchanging the data words D1, intially in PEM, and D2, initially in mass memory (MM). Both words contain 512 bits and D1 is stored across one PE row: row j (PEi in row j contains the $i^{\underline{th}}$ hexadecimal digit of D1). Recalling the IOBR structure presented in Figure 22, the general procedure is the following:

clock 0 - Initiate a MM read of word D2 to IOBRr.

clock 8 - Initiate a PEM read of word D1.

clock 10 - MM read is completed and D2 is in IOBRr. The PEM read will be completed during the next clock period, therefore gate D1 through row-gating to IOBRr and simultaneously shift IOBRr left 128 digits (i.e., IOBRℓ ← IOBRr). This can be done in one clock.

clock 11 - At this instant, IOBRr contains D1 and IOBRℓ contains D2 Initiate now the MM rewriting which will replace D2 by D1 in MM. Also initiate a PEM write which will write D2 from IOBRℓ into any PEM row selected by row gating. If the row selected is row j, then D2 will replace D1 in

that PEM row.

clock 16 - PEM write is complete; D2 is now available in PEM.

clock 21 - MM rewrite is finished; ready to start a new I/O transaction at this clock.

One elementary I/O transaction then takes: 1 MM cycle and 1 PE clock or approximately one MM cycle, which was assumed to be 2 μsec (1 μsec access time, 1 μsec rewrite time). Eight of these elemtnary I/O's are needed to exchange one digit in every PEM with MM since there are eight PE rows. Therefore:

$$T_{I/O} = 168N \qquad (13)$$

where $T_{I/O}$ is the time in clocks to interchange a word N digits long between PE's and MM. For N=8, $T_{I/O} \cong 135$ μsec. This indicates that since a typical 32-bit floating-point operation takes about 25 μsec, each word brought to PEM should be used on at least six operations (before being overlaid to MM) in order to completely overlap execution and I/O.

The procedure described above for I/O transactions is based on the assumption that MM is bulk core. In this case, IOBRr is in fact the memory data register for MM. If MM is implemented with semiconductor memory, then it would be better to modify the structure in Figure 22 and have the output data from MM linked to IOBRℓ and the input data linked to IOBRr. This would avoid the IOBR shift in clock 10 and would save one clock in each transaction.

4.7 Routing

The following algorithm is employed to perform routing left of one digit by a distance R, R $\leq$ 1023. This is obviously general since a routing right by n is equivalent to a route left by 1024-n.

1) IOC, which processes routings, decomposes R into r'=R div 128

and r=R mod 128. r' will be taken care of by row gating and r by shifting IOBR.

2) IOBRr is loaded with row r' from PEM (rows are numbered from 0 through 127).

3) IOBRr is shifted left 128 thus placing row r in IOBRℓ; simultaneously row r'+1 is brought to IOBRr.

4) IOBR is shifted left by a distance r.

5) IOBRℓ now contains the routed word for row 0. Therefore, IOBRℓ is written into row 0.

6) IOBR is now shifted by (128-r) which places row r'+1 into IOBRℓ simultaneously, row r'+2 is brought to IOBRr

7) Repeat step 4.

⋮

and so on

It should be noticed that row r' has to be brought to IOBR twice, once at the beginning and once at the end of the routing. This is necessary to recover the leftmost digits of r' which are lost when step 4 is first executed.

The actions performed are: 9 row loads into IOBRr, 1 shift by 128, 8 shifts by r, 7 shifts by (128-r) and 8 stores of IOBRℓ into rows. Also the first clock of all but the first row loads is overlapped with the last clock of a shift and the first clock of all but the last IOBR stores is overlapped with the first clock of a shift. Therefore, the timing for routing will be given by:

$$T_r = t_\ell + 8(t_l-1) + 8t_{sh(r)} + 7t_{sh(128-r)} + t_{sh(128)} + 7(t_s-1) + t_s$$

where T_r is the time in clocks for routing one digit by a distance R = 128r'+r;

t_l is the number of clocks for a row load; $t_{sh(r)}$ is the number of clocks to shift IOBR by r; and t_s is the number of clocks for a row store. It is known that $t_{sh(128)}=1$. The values for t_s and t_ℓ depend on where the digit to be routed is: if it is in some PE register, then these times are only one clock; if the digit is in PEM, then t_ℓ requires one PEM read or 3 clocks and t_s takes 5 clocks for a PEM write. Therefore, there are four different types of routing. They are, from the fastest to the slowest: 1) PE to PE, 2) PEM to PE, 3) PE to PEM and 4) PEM to PEM.

For routings of type 1:

$$T_{r1} = (8t_{sh(r)} + 7t_{sh(128-r)} + 3)N \tag{14}$$

where T_{r1} is the time in clocks to route a number with N digits. $t_{sh(r)}$ is given by Table 9 for $r \leq 64$; shifts by $r > 64$ in a given direction are simply obtained by first shifting by 128 (end around) and then shifting (128-r) in the opposite direction. $t_{sh(r)}$ for $r > 64$ can thus be written as $1+t_{sh(128-r)}$ and $t_{sh(128-r)}$ is taken from Table 9.

For N=8 and r=1, one obtains T_{r1} = 20 μsec. This is the best possible routing time and it is on the order of one floating-point operation time. Other distances may take longer. For example, when N=8 and r=2, T_{r1} is 32 μsec. Note also that routing must always be from one location to another or else the row that must be loaded twice would be changed when accessed for the second time.

For routings of types 2, 3, and 4 the expressions are:

$$T_{r2} = (8t_{sh(r)} + 7t_{sh(128-r)} + 21)N \tag{15}$$

$$T_{r3} = (8t_{sh(r)} + 7t_{sh(128-r)} + 35)N \tag{16}$$

$$T_{r4} = (8t_{sh(r)} + 7t_{sh(128-r)} + 53)N \tag{17}$$

It is also important to notice that since routing is performed in chunks of 128 each, several other special purpose types of partial routings can be microprogrammed and are very useful in specific applications.

4.8 Summary of Timings

Table 11 presents a summary of the timing estimates for several operations and four "typical" word lengths: 16 bits (N_m=3, N_e=1), 32 bits (N_m=6, N_e=2), 48 bits (N_m=9, N_e=3), 64 bits (N_m=12, N_e=4).

Operation	Formula Number	Time in μsecs			
		16 bits	32 bits	48 bits	64 bits
Local indexing, per address	---	1.6	1.6	1.6	1.6
Mantissa multiplication	1	12	39	82	141
Floating-point multiplication	2	9.4	26	52	86
Fixed-point unsigned addition	3	4.8	9.6	15	19
Fixed-point signed addition	4	7.4	14	20	27
Floating-point addition	5	12.5	20	28	35
Mantissa division	6	44	145	na	na
Logic Operations	7	4.8	9.6	15	19
Comparison of unsigned numbers	8	2.8	5.6	8.4	11
Comparison of signed numbers	9	3.8	6.6	9.4	12
Comparison of floating-point numbers	10	4.8	7.6	11	13
Global shifts	11	4.8	9.6	15	19
Locally indexed shifts	12	6.8	12	17	21
I/O (PEM←→MM)	13	67	135	200	269
Routing PE - PE, distance 1	14	10	20	30	40
Routing PEM - PE, distance 1	15	17	35	52	69
Routing PE - PEM, distance 1	16	23	46	69	91
Routing PEM - PEM, distance 1	17	30	60	90	120

Table 11. Summary of Timing Estimates

5. APPLICATIONS

5.1 General Considerations

In general, SPEAC can handle efficiently most problems in which ILLIAC IV performs well since most of the features of ILLIAC IV are also available in SPEAC. A large number of parallel algorithms to implement many important applications in ILLIAC IV have been developed [9 through 17]. Obviously, these algorithms can be used as a starting point when the use of SPEAC for the same applications is contemplated. A few modifications or a new approach are sometimes required due to the following differences:

a) PEM is much smaller in SPEAC and many problems which are "core contained" in ILLIAC IV must use memory overlay in SPEAC. On the other hand, MM in SPEAC is random-access and the machine was especially designed to allow efficient PEM overlay so it is normally possible to use SPEAC efficiently even in non-core contained problems. In ILLIAC IV, non-core contained problems, while not as frequent as in SPEAC, are harder to program efficiently due to the latency problem in its disk mass memory.

b) Routing is relatively slow in SPEAC. While in ILLIAC IV a route takes about half the time required for a floating-point operation regardless of distance, in SPEAC it takes from one to several times as much as a typical floating-point operation, depending on the distance. On the other hand, in SPEAC routing is an I/O operation and can be overlapped with PE processing. Also special route instructions can be microprogrammed, "customized" to particular problems.

c) ILLIAC IV is primarily intended for computations on floating-point numbers with 32 or 64 bits precision. While SPEAC can also handle these problems, floating-point multiplication becomes relatively slow for very long word lengths since it is proportional to the square of the number of digits in the word. Furthermore, there is a very important area of applications which is much more "natural" to program for SPEAC than for ILLIAC IV. This area includes problems involving a large quantity of fixed-point numbers with small precision, typically only a few bits. Examples of these problems are: picture processing, non-numerical processing in strings of characters, etc. These problems can be handled very efficiently by SPEAC due to its digit-by-digit processing and fast operation for small words.

d) In ILLIAC IV, the number of PE's (n_{PE}) is 64 and for most applications one is interested in tackling problems in which the number $\underline{n}$ of parallel computations is equal to or greater than n_{PE}. In matrix computations, for example, n is the order of the matrix and in discrete Fourier transforms n is the number of points. Therefore, a frequent problem in ILLIAC IV is to partition a large data set into "chunks" of 64 or 64×64 so that each chunk can "fit" in the machine. Chunks are then processed sequentially. In SPEAC, n_{PE}=1024 and for most problems one will be interested in $n \leq n_{PE}$; the typical problem is to subdivide a data set into several pieces and to process all the pieces in parallel to "fill" the whole machine when $n < n_{PE}$.

In the next sections a few specific representative applications of

SPEAC are considered in detail. Of course, they are only meant as a sample since many other interesting applications could possibly be efficiently handled by the machine.

Timing estimates were based on counting PE clocks by hand. Some attempt has been made to take into account PE/IO overlap but precise numbers could only be obtained with a very sophisticated simulator for CU and specific detailed microprograms for every instruction. Therefore, the estimates can be a little pessimistic if the overlap was not fully accounted for.

5.2 Relaxation

The problem consists of: given an initial matrix U^0, $n \times n$, find a succession of matrices U^1, U^2, ... where each term of matrix U^{k+1} is a function of the four "neighbors" of the term in the previous matrix U^k.

In general,

$$U_{ij}^{k+1} = f(U_{i+1,j}^k, U_{i-1,j}^k, U_{i,j+1}^k, U_{i,j-1}^k, U_{i,j}^k)$$

This is a general formulation for a series of problems that can be very efficiently solved using an array computer. If the elements of U are floating-point numbers, then this type of expression can be used to find the equilibrium temperatures or potentials at every point of a plane submitted to given initial conditions at the edges; if the elements of U are small integers, then each element can represent a point of a picture coded according to a gray scale. In this case, the formulation can be used to implement a "smoothing" filter or a number of other picture processing problems.

As an example, the following case will be studied.

$$U_{ij}^{k+1} = \frac{U_{i+1,j}^k + U_{i-1,j}^k + U_{i,j+1}^k + U_{i,j-1}^k}{4}$$

The loop condition is the following: if $|U^{k+1}_{i,j} - U^{k}_{i,j}| < \epsilon$ for all i,j then exit the loop; otherwise repeat.

Two values of n are considered: 32 and 1024 although other powers of two can also be handled efficiently.

a) n=32; the elements of U are 32-bit floating-point numbers. The most straightforward (and most inefficient) way of coding the loop is: the elements of U are stored across PE's, row after row; i.e., numbering PE's from 0 to 1023 and rows from 0 to 31, element U_{ij} is stored in PE_{32i+j}. U^0 is in PEM location a. The loop is:

1) Route distance 1 left from PEM location a to PEM location b.
2) Route distance 1 right from PEM location a to sM(0).
3) Add sM(0) to PEM(b) and store in PEM(b).
4) Route distance 32 left from PEM(a) to sM(0).
5) Add PEM(b) ← (PEM(b)+sM(0)).
6) Route distance 32 right from PEM(a) to sM(0).
7) Add sM(0) ← (PEM(b)+sM(0)).
8) Multiply the addition of the four neighbors by .25, sent via CDB:
   sM(0) ← (sM(0) × CDB(.25)).
9) Test for ending condition; sM(0) which now contains U^{k+1} is subtracted from U^k which is in PEM location a and the difference is compared against ϵ, sent via CDB. In PE's in which the ending condition is satisfied, a zero is gated to the interrupt wire and register M is reset which disables the PE.
10) Write sM(0) in PEM location a and go back to step 1.

The process ends when, in step 9 CU receives a zero via the interrupt wire; this indicates that all PE's are disabled. At this point the result

of the last iteration is stored in sM(0); all PE's are enabled and the result can then be stored in PEM. The procedure requires three additions (20 μsec), one subtraction (20 μsec), one multiplication (25 μsec), three routes of type 2 (35 μsec), one route of type 4 (60 μsec), and one comparison (7.6 μsec) for a total of 278 μsec per execution of the loop. Obviously, each execution of the loop computes a new iteration matrix--U^{k+1} out of the previous value U^{k}. It should be noticed that sM was used as temporary storage in some steps. sM can store two 32-bit numbers: one in sM(0) through sM(7) and the other in sM(8) through sM(15). It is obviously possible to write microsequences for variants of addition and multiplication which take one or both operands from sM instead of PEM and also possibly have the results in sM instead of storing the numbers back in PEM. These operations will be faster than the normal PEM to PEM ones (from 1.6 to 6.4 μsec faster) but this will not normally be taken into account in these worst-case timings. It is also important to notice that since sM is used as scratchpad in most operations, if the two operands are in sM, one is destroyed during the operation unless sM is enlarged to contain four or eight 32-bit numbers instead of only two.

A few improvements are possible in the straightforward algorithm presented above and they are as follows:

1) The routing in step 1 does not have to be of type 4 since sM is available. Therefore, one can load the data in sM(0) (2.4 μsec), route from sM(0) to sM(8) (20 μsec), and store sM(8) in PEM (4 μsec). These last four μsec can be overlapped with the routing and total time is roughly 25 μsec.

2) A special microsequence can be written for an instruction to divide by 4 by shifting and normalizing. This will take much

less than 25 μsec; since the operand is in sM and the result is also left in sM, 5 μsec is a reasonable upper bound.

3) All additions except the last can be overlapped with routings of type 2. The routings to be overlapped must be of type 2 because there is no space in sM to keep the elements of U^k permanently in sM, which would enable one to use only type 1 routings. If more space were available in sM, the sum could also be kept in sM and PEM location b would not be used.

The improved algorithm is as follows:

1) sM(0) ← PEM(a); U^k is now in sM(0) (2.4 μsec).

2) Route distance 1 left from sM(0) to sM(8) (20 μsec). Simultaneously, write sM(8) in PEM(b) (~2.6 μsec).

3) Route distance 1 right from sM(0) to sM(8) (20 μsec).

4) PEM(b) ← (PEM(b)+sM(8)). Simultaneously, route distance 32 left from PEM(a) to sM(0) (35 μsec).

5) PEM(b) ← (PEM(b)+sM(0)). Simultaneously, route distance 32 right from PEM(a) to sM(8) (35 μsec).

6) sM(0) ← (PEM(b)+sM(8)) (~16 μsec). Note that this addition takes less time because the result is not stored back in PEM.

7) sM(8) ← sM(0) shifted 2 right (i.e., divided by 4) and normalized (~5 μsec).

8) Test end condition. This is the same as step 9 in the original algorithm (~8 μsec).

9) PEM(a) ← sM(8). Go to step 1 (~4 μsec).

The total time is now only ~148 μsec for each complete relaxation. Further improvement is possible if sM can store four 32-bit numbers instead

of only two. In this case all routings are of type 1 (which saves 30 μsec) and the whole problem can be done in sM which saves all PEM reads and writes except the initial read and final write. In this case a total time on the order of 110 μsec is possible.

The algorithms considered assume a toroidal geometry; i.e., there are no edges, $U_{0,j}$ is considered a neighbor of $U_{31,j}$ and $U_{i,0}$ a neighbor of $U_{i,31}$. This is not desirable for most actual applications. In most cases, there is an outside edge: $U_{-1,j}$, $U_{32,j}$, $U_{i,-1}$ and $U_{i,32}$ with fixed values. This can be easily included in the program in the following way: a digit D is stored in each PE containing the LSB ON if the element stored in that PE belongs to row 0, the second LSB ON if it belongs to row 31, the third LSB ON for column 0, and the MSB ON for column 31. The fixed edge values are stored in PEM locations c,d,e and f (each is only needed in 32 PE's, but it is probably easier to store them in all PE's). A new step is needed between 2 and 3 in the improved algorithm. This step is number $2\frac{1}{2}$ and is identical with step 1. Before steps 1, $2\frac{1}{2}$, 4, and 5, a local indexing is added. This local indexing is enabled by the bits of D and makes PE's that have an edge neighbor take the edge value instead of the "end-around" neighbor. This adds only about 8 μsecs to the procedure.

It should also be pointed out that overlaps of two operations both using sM can be less than perfect since sM has only one port. Normally, however, operations that use sM do so 50% or less of the clocks and thus very good overlap is possible. Multiplication is an exception since it uses sM very heavily.

b) n=1024; the elements of U are floating-point 32-bit numbers.

In this case each row of U is stored across PE's and 1024 rows are needed. Therefore, the problem is not "core contained" and PEM overlay is necessary. Routing is only needed now to access the "left" and "right" neighbor; the "upper" and "lower" neighbors of an elements and the element itself are stored in the same PE. Therefore, at least three complete rows of U must always be present in PEM. Assuming they are in locations a, b, and c respectively, the algorithm is:

1) sM(0) ← PEM(b); U^k is now in sM(0) (2.4 μsec).

2) PEM(d) ← (PEM(a)+PEM(c)); do not destroy sM(0) (20 μsec).

3) Route distance 1 left from sM(0) to sM(8) (20 μsec).

4) PEM(d) ← (sM(8)+PEM(d)); do not destroy sM(0) (20 μsec).

5) Route distance 1 left from sM(0) to sM(8) (20 μsec).

6) sM(0) ← (sM(8)+PEM(d)) (~16 μsec).

7) sM(8) ← sM(0) shifted 2 right and normalized (~5 μsec).

8) Test end condition (~8 μsec).

9) PEM(b) ← sM(8); go to step 1 (~4 μsec).

Steps (2,3) and (4,5) could overlap for a total time of 58 μsec per row. However, this would leave only 5.7 μsec in which both buses are not simultaneously used and I/O overlay could not occur. Since FINST normally assigns priority to I/O, on the average each loop will take the maximum time of 116 μsec and will have to wait for 20 more μsec for I/O. Therefore, the procedure is I/O bound and each loop takes 135 μsec which is the time needed for an I/O transaction. One iteration is then performed in about 135 msec. Fixed edge conditions can be introduced as discussed in case a and do not cost any extra time since the procedure is I/O bound.

c) n=1024; the elements of U are one-digit integers. This case would be used in picture processing. The problem is "core contained" since 2K digits are available in PEM and only 1K are needed. Storage is as in case b, each row across PE's. All elements of the same column are in the same PEM. Only one PEM read and one PEM write are needed per row since sM is now capable of storing sixteen 4-bit elements. Assume that sM(a) contains the upper neighbor, sM(b) the present element and sM(c) will contain the lower neighbor. The algorithm is:

1) sM(c) ← PEM(address of element of next row).

2) reg A ← sM(a)+sM(c).

3) Route distance 1 left from sM(b) to sM(d) (2.5 μsec).

4) reg A ← reg A+sM(d) (.2 μsec).

5) Route distance 1 right from sM(b) to sM(e) (2.5 μsec).

6) reg A ← reg A+sM(e) (.2 μsec).

7) Shift reg A right 2 bits (.2 μsec).

8) Test end condition (~.5 μsec).

9) Got to step 1.

The whole procedure then takes only about 6 μsec since steps 1, 2, and 4 are overlapped with routes. Therefore, one iteration can be performed in about 6 msec. Fixed edge conditions could be introduced without difficulty since there is space in sM to keep the data for the edges. This problem could also use two digits per element for a gray scale with 256 shades. Since sM can still be used, the time increases linearly to 12 msec per iteration.

In conclusion, SPEAC performs exceedingly well in relaxation type problems.

5.3 Matrix Multiplication

Given two matrices, $A_{n\times n}$ and $B_{n\times n}$, the problem consists of finding the matrix $C_{n\times n}$ which is the product of A and B. $C=A\times B$ $(c_{ij}=\sum_{k=1}^{n} a_{ik} \times b_{kj})$.

Two basic methods can be used to store matrices in an array computer:

a) Straight storage, in which each row is stored across PE's and all elements of a column are stored in the same PE. Therefore, a_{ij} is stored in PE_j.

b) Skewed storage, in which each row is stored across PE's but it is also rotated one position farther than the preceeding row in an end-around fashion. Thus, a_{ij} is stored in $PE_{(i+j-2) \bmod n+1}$.

In either storage scheme one row of A can be accessed by fetching one row of PE memory. When a matrix is skewed one column can also be accessed in one memory fetch by indexing each PE to a different memory location. To fetch the first column of A, for example, each PE simply loads from location A plus the number of that PE. By routing this indexing pattern, any column of A can be accessed in one operation. It would take many memory fetches to access a column of a matrix which is not skewed since all elements of a column are stored within one PE.

Three methods have been proposed ([11] and [12]) to perform matrix multiplication in an array computer. Briefly, they are as follows:

a) the log-sum method, which is used to multiply skewed matrices since columns and lines must be accessible. A row of the first matrix is fetched and multiplied, in parallel, by a column of the second matrix. The results are summed across PE's to produce one

element of the solution. There are two major causes of inefficiency in this method. First of all, the operation of summing across PE's, known as a log-sum, is at best only 20% efficient in using PE's. Secondly, excessive routing is required to properly index columns and line them up with rows.

b) the broadcast method which generates one row of the result matrix at a time rather than just one element. It operates on matrices which are stored straight in memory and produces a result matrix which is also stored straight. Each row of the result is obtained after n multiplications and accumulations (the result of each multiplication is added to the sum of all previous multiplications). To obtain row i of the result, the k^{th} element a_{ik} of row i of matrix A is multiplied by the k^{th} row of matrix b and all n rows thus obtained are added together. The expression is:

$$\text{row}(c_i) = \sum_{k=1}^{n} a_{ik} \, \text{row}(b_k)$$

The CU must be able to broadcast the elements a_{ij} to the PE's and the PE's must have access to rows of B (i.e., row across PE's). As opposed to the skewed matrix multiplication, this method is almost 100% efficient. There is no log-sum involved and no routing is required.

c) Knapp's method of which only a brief description is offered here; for a detailed treatment see [12]. A and B are stored straight and C will also be obtained straight. As in the broadcast method, each row of the result is obtained after n multi-

plications and accumulations. However, no broadcast takes place. To obtain row i of the result, row i of A is multiplied by each diagonal of B and then routed right one. Defining the $k^{\underline{th}}$ diagonal of B as:

$$b_{1,k},\ b_{2,k+1},\ \cdots,\ b_{i,(k+i-2)\bmod n+1},\ \cdots,\ b_{n,(k+n-1)\bmod n+1}$$

then Knapp's method is expressed by the following:

$$\text{row}(c_i) = \sum_{k=1}^{n} (\text{row } a_i \text{ routed right } (k-1) \text{ times}) \times (k^{\underline{th}} \text{ diagonal of B})$$

To access the first diagonal of a matrix stored straight, each PE is locally indexed with the PE number (starting with 0); this pattern is routed right (k-1) times to access the $k^{\underline{th}}$ diagonal. The efficiency of Knapp's method is very good because no log-sum operations are performed, but not as good as straight multiplication since routing is required. Its major use is to perform several small matrix multiplies simultaneously using only a small group of PE's for each one.

The three methods can be used in SPEAC but the log-sum method is not considered in detail since it is the least efficient. Two cases are studied:

a) n=1024 and each element is a 32-bit floating-point number. Each matrix is stored straight and the broadcast method will be used. One slight modification is needed, however, to avoid I/O bounding since the problem is not "core contained." In the broadcast method the rows of B are used in order from row 1 to row n (to compute row 1 of C) and then again from row 1 to row n (to compute row 2 of C) and so on. Therefore, each row is used only for one multiply and add each time it is in PEM. Since n multiply and adds can be

performed in 45 μsec, there is no time to overlay a row which takes 135 μsec. The solution is simple; each row of B must be used several times each time it is brought to PEM. In this way, several rows of the product are computed simultaneously. For example, the first row of B (row(b_1)) is brought and is multiplied by 64 broadcast elements $a_{1,1}$, $a_{2,1}$, ··· , $a_{64,1}$. The 64 rows thus obtained are stored in PEM. Row(b_2) is then accessed and is multiplied by $a_{1,2}$, $a_{2,2}$, ··· , $a_{64,2}$; each of the 64 rows thus obtained is added to the corresponding row of the first 64. At the end of 1024 cycles, all rows of B have been accessed and used 64 times each, and the first 64 rows of C are completed. The method is repeated sixteen times to obtain the 1024 rows of C. Since each multiply and add takes 45 μsec, 64 take 2880 μsec in which there is time to interchange 21 rows. Therefore, I/O can be easily overlapped with execution; while the 1024 rows of B are used, there is time to interchange 21K rows and all that is needed is to interchange 1024 rows plus the 64 result rows.

CU obtains the elements to broadcast either directly from mass memory or from the PE's via CDB. The latter is the most straightforward scheme and can be efficiently used since overlap is possible with execution due to the fact that I/O takes a relatively small percentage of the execution time. 64 rows of A are needed in the PE at all times to obtain the broadcast elements. Patterns are also stored in PEM and used to turn off all but one PE each time CDB_{out} is used to send a broadcast element to CU. Note also that CU can simultaneously broadcast a previous element since CDB_{in} is used for this purpose.

In the worst case, there are 194 rows in PEM at one time: the 64 rows of C that are being computed, the 64 rows of C that have just been

completed and have not been placed in MM yet, 64 rows of A that are being used to obtain broadcast elements, and 2 rows of B--one being used to multiply and a new one being prepared for the next step. When the 64 completed rows of C are overlaid to MM, the space is used to load the next group of 64 rows of A. When a new step begins, the locations of the old rows of A are used to place the new partial rows of C.

Therefore, complete overlay of I/O and CU instructions is possible and the timing is simply given by: n^2 (multiplications and additions). sM can contain the row of B being used 64 times and also the result of the multiplication. Only the result of the addition must be stored. In these conditions, multiply and add takes about 43 μsec and the final result is 43 sec.

b) $n=1024/2^k$ (k=1,2,3,4) and each element is a 32-bit floating-point number. This is the submultiple case, in which the size of the matrix is a submultiple of the size of the array. In order to keep all PE's busy, one can either divide the matrix in $\frac{n_{PE}}{n}$ parts and use all PE's to compute one multiplication or $\frac{n_{PE}}{n}$ multiplications can be computed simultaneously. The two approaches are very similar and only the first is considered. Two methods can be used; the broadcast method, which is especially suitable when $\frac{n_{PE}}{n}$ is small (2 or 4 ideally) and Knapp's method which is best when $\frac{n_{PE}}{n} \gg 8$.

In the broadcast method, $\frac{n_{PE}}{n}$ repetitions of a row of B can be catenated across one row of PEM's and the method is used as before but instead of generating k rows of C at the end of each step (k=64 in the example presented in part a), $k \times \frac{n_{PE}}{n}$ rows of C are constructed simultaneously. For $\frac{n_{PE}}{n} \leq 8$ this repetition is easily obtained by writing in PEM $\frac{n_{PE}}{n}$ times the same row of B read only once from MM. Obviously, there is one difficulty:

the broadcast element must be different for each of the $\frac{n_{PE}}{n}$ copies of the row of B. Up to four different broadcast elements may be sent during a multiplication of two digits without any extra delay. The only problem is to enable sM's in only a portion of the PE's without disabling the multiplication itself. This suggests the introduction of an enable flip-flop for I/O and CU use and sM may be directed to obey either the PE enable or the I/O/CU enable. If this is available, the broadcast method can be used without any extra cost since the multiple broadcasts are overlapped with multiplication. Therefore: time = $43 \frac{n^2}{n_{PE}/n} = \frac{n^3}{n_{PE}} \times 43$ μsec, and for a 256 × 256 matrix the time = 740 msec.

If the above mentioned control of sM is not available, about $8(\frac{n_{PE}}{n}+2)$ additional clocks are needed per multiplication to select the broadcast elements. For $\frac{n_{PE}}{n} = 4$, this adds 5 μsec per multiplication. The expression is: time = $\frac{n^3}{n_{PE}} \times (43 + .8(\frac{n_{PE}}{n} + 2))$ μsec.

This method is then convenient only when $\frac{n_{PE}}{n}$ is small so that the extra time spent in selective broadcast is not excessive.

Knapp's method avoids selective broadcasts but introduces routings. $\frac{n_{PE}}{n}$ rows of A are concatenated across one row of PEM's and B is repeated $\frac{n_{PE}}{n}$ times, once for each concatenated row of A. For n=128, this operation is easily obtained by writing in PEM eight times the same row of B read only once from MM. For $n < 128$ this repetition may require initial routes. Each diagonal of B is obtained by local indexing ($\frac{n_{PE}}{n}$ copies of the diagonal are actually obtained) and multiplied by the rows of A. The result is accumulated and when all diagonals have been used, $\frac{n_{PE}}{n}$ rows of C are computed. After each diagonal is used, the rows of A must be routed right by a distance of 1. Since this route is end-around with respect to n and not to n_{PE}, a second route is needed (by a distance n) unless n=128. The rows of A can be kept in sM while

in use so routing is of type 1. The time is given roughly by the following:

$$\text{time} = \frac{n^3}{n_{PE}} \text{ (add time + multiply time + 2 route times)} \cong 90 \frac{n^3}{n_{PE}} \mu\text{sec}$$

if the routes are all fast. Therefore, the selective broadcast method is best for all cases in which $\frac{n_{PE}}{n} \leq 32$.

5.4 Pattern Matching

This application was chosen to test the character manipulation capabilities of SPEAC. The problem, fully described in [9], is briefly stated as: given two strings of characters, S (with n_s characters: $s_1, s_2, \ldots, s_{n_s}$) and P (with n_p characters: $p_1, p_2, \ldots, p_{n_p}$), find out how many times and/or in which position does P occur in S. P is called the pattern string and S the source string. Normally $n_s \gg n_p$. The problem can be considered in two different aspects: 1) n_s is very small (typical 1 to 3) and only the count of occurrences is desired. This is what is needed in analysis of texts to obtain the frequency of occurrence of given letters or combinations of letters, and 2) n_s can be a small integer up to about 15 and the positions in S in which P occurs are desired. This is the type of algorithm needed, for example, to find all occurrences of the words BEGIN and END in a segment of a program as would be necessary in a parallel compiling technique as proposed in [10].

The source string S can be arranged in memory in two different ways: 1) S is distributed across PE's in rows, one element per PE; i.e., character S_i is in $PE_{(i \bmod n_{PE})}$, and 2) S is distributed across PE's in n_{PE} chunks each with n_s/n_{PE} adjacent characters; i.e., character S_i is in $PE_{(i \text{ div } n_s/n_{PE})}$. Storage scheme 2, called storage in chunks, leads to much more efficient programs in SPEAC than storage scheme 1, called storage across PE's. This is due

to the fact that with storage in chunks, routing is practically eliminated. However, both storage schemes are considered since it may be difficult to use storage in chunks if the input data is not initially manipulated by corner memory.

a) Storage in chunks; only a count of the number of occurrences is required. Each character is assumed to be four bits long and is coded in one digit. Obviously, this introduces no restriction since the same algorithms can be applied if more than one digit is needed to code each character. No character manipulation instructions were considered in Chapter 4. Therefore, most instructions used in these algorithms are custom-made, that is, they are described in terms of their microsequences.

Initially, the first (n_p-1) characters in each chunk must be routed left by a distance of one in order to enable the recognition of truncated occurrences of P (i.e., an occurrence of P in which p_1 is the right-most character in chunk i and p_2, p_3, ... , p_{n_p} are the first characters in chunk i+1). The initialization thus takes (n_p-1) routings distance 1 or $(n_p-1) \times 2.5$ μsec.

Ideally, for best efficiency, the length n_c of each chunk $(n_c=n_s/n_{PE})$ is a large number. n_c is here considered to be on the order of 1K; i.e., the source string has one million characters. If S is longer, the whole procedure is repeated a number of times; each execution analyzes one million characters.

The following algorithm can be used: X_1 contains the address of the next character in S to be analyzed. The pattern string is initially brought to CU and will be repeatedly broadcast via CDB.

1) X_1 is loaded via CAB with the address of the next character of S

to be analyzed as a possible start of an occurrence of P: $X_1 \leftarrow$ CAB(address of S_i) (1 clock). Simultaneously, all lcFF1 are turned ON: lcFF1 $\leftarrow$ ON via CDB.

2) Compare the characters of S and P and turn off PE's in which no match is found: $A_m \leftarrow PEM(X_1)$; lcFF1 $\leftarrow (A_m = CDB(p_j))$; Increment X_1; Enable function is attributed to lcFF1 ON (4 clocks).

3) Step 2 is repeated n_p times, for j=1,2, ... , n_p. At the end of this loop, lcFF1 is ON only if there was a match.

4) Count the match by incrementing A_c in PE's in which there was a match. A_c is initially zero; increment A_c, enabled by lcFF1 ON (1 clock).

5) Go to step 1. The whole procedure is repeated $n_c = n_s/n_{PE}$ times, using as S_i: s_0, s_1, ... , s_{n_c}.

6) At the end of the chunk, A_c contains the number of matches in each PE; no overflow is possible since A_c can store up to 4K and only 2K matches are possible if $n_c = n_{c\ maximum} = 2K$. A log-sum of the contents of A_c is then performed and the final total may be sent to CU via CAB.

The kernel in the algorithm above can now be timed; step 2 is repeated n_p times and the loop is repeated n_c times, for a total of $n_c(4n_p+1)$ clocks. The initialization takes $20(n_p-1)$ clocks and the finalization takes ten routings of type 1 and ten additions of 16-bit unsigned numbers for the log-sum, for a total of 150 clocks. Therefore:

$$\text{Total time} = (n_p-1)20 + n_c(4n_p+1) + 150 \text{ clocks.}$$

For n_c=1K and n_p=5, the total time to search one million characters for a match

is only 2.1

b) Storage in chunks; the location of each occurrence is required. The algorithm is very similar to the one in case a, but now X_2 is also used to hold the location in PEM where the address of the next occurrence will be stored. Step 4 is replaced by the following:

4) Store the occurrence of the match in each PE by writing the address of S_i, the first character of the occurrence, in X_2; X_2 is then incremented by one. Since the whole step is enabled only in PE's in which there was a match, each list of occurrences is compact, with no vacant locations: PEM(X_2) ← CDB(address of S_i); Incr X_2; attribute enable function to lcFF1 ON. Three PEM writes are needed since an address has three digits.

The new step 4 takes 12 clocks and the new total time is:

Total time=$(n_p-1)20 + n_c(4n_p+12)$ + 150 clocks.

For n_c=1K and n_p=5, the total time is now 3.2 msec.

c) Storage across PE's; only a count of the number of occurrences is required. Since in this storage scheme adjacent digits are in adjacent PE's, left routings of distance 1 are needed between comparisons. There is also a problem with the right-most PE's; at a routing, these PE's should receive characters from the next row of characters rather than end-around characters from the present row. For each row of characters, the algorithm is as follows:

1) Load in A_m the characters of the old next row (the present row) which are in sM(0): A_m ← sM(0) (1 clock).

2) Fetch the next row of characters from PEM and store in sM(0):

$sM(0) \leftarrow PEM(X_1)$ where X_1 contains the address of the characters in the next row (3 clocks). Simultaneously, all lcFF1 are turned ON via CDB.

3) Compare character in A_m with the first character of P, sent via CDB; the result is stored in lcFF1, enabled by lcFF1 ON: lcFF1 $\leftarrow (A_m = CDB(p_1))$ (1 clock).

4) Store A_m in B to prepare for the routing: $B \leftarrow A_m$ (1 clock).

5) Replace B by sM(0) only in the first PE. In this way, B will contain the row needed for routing. $A_m \leftarrow sM(0)$ enabled only in the first PE (2 clocks).

6) Route 1 character left, distance 1 from B to A_m (20 clocks).

7) Same as step 3 but using p_2.

8) Repeat steps 4 through 7 (n_p-1) times. For the $i^{\underline{th}}$ execution, character p_{i+1} of P is used and the first $\underline{i}$ PE's are enabled in step 5. Therefore, p-1 different patterns are needed to enable PE's in step 5. Since p is small and each pattern takes only 1 bit per PE, these patterns may be stored in sM and enabling takes only 1 clock.

9) lcFF1 is now ON only if a match occurred; A_c is incremented to store this fact: Incr A_c enabled by lcFF1 ON (1 clock).

The whole algorithm is repeated once for each row. At the end of the procedure, a log-sum of A_c is performed to obtain the total number of occurrences. This takes 150 clocks. The total time to process n_r rows is then:

$$\text{Total time} = n_r(6 + 24(n_p-1)) + 150 \text{ clocks.}$$

To analyze one million characters when n_r=1K and n_p=5, 10.2 msec are required.

Therefore, this algorithm is about five times slower than the one for chunk storage.

d) Storage across PE's; the location of each occurrence is required. Only a small modification is needed in the algorithm of case c, similar to the modification introduced in case b. Instead of using A_c to keep the number of matches, X_2 is used to keep the address in PEM where the address of the next match will be stored. This step adds 12 clocks per row to the algorithm of case c, thus yielding a total time of:

$$\text{Total time} = n_r(17 + 24(n_p - 1)) + 150 \text{ clocks.}$$

Or, for 1K rows and $n_p = 5$, 11.3 msec.

Therefore, pattern matching can be performed very efficiently in SPEAC. One final sophistication to improve performance if the number of occurrences is small is the following: when testing for each possible match, gate lcFF1 to the interrupt wire after each comparison. If the CU receives a zero, this means that that match failed in all PE's and the present attempt can be abandoned without testing all the remaining digits of P. This step costs no extra time and could provide an impressive improvement for large values of n_p (i.e., $n_p > 10$).

5.5 Sparse Matrices

The problem deals with the elimination of the need to store in PEM the zero elements of sparse matrices and the resulting problem of remembering in some form the positions of the non-zero elements in the actual matrix. The term actual matrix will be used to refer to a sparse matrix represented with its zeroes and actual row to refer to a row of such a matrix also with its zeroes. The form decided upon clearly must be useful in completing the task

of sparse matrix multiplication. This section is concerned with describing two forms of storing sparse matrices for SPEAC, discussing their program adaptability, and demonstrating their use in programming.

The two general forms for storing sparse matrices are the individual-tag method and the bit-matrix method [11]. These two methods are similar in that for both, the non-zero elements of a matrix are stored in the same way; for a sparse matrix A, 1024×1024, the $j^{\underline{th}}$ column is stored in PE_j and zeroes are eliminated by pushing each non-zero number up the column until no zero elements remain between it and the next higher non-zero element, if one exists.

1) The bit-matrix method consists of storing a 1 or a 0 bit for each element of the actual matrix depending on whether an element is non-zero or zero respectively. The result of this procedure is a matrix with the same dimensions as the actual matrix, but which requires less space to store in memory since each element of this matrix is only a bit wide. These bits are stored packed four in each digit and require 256 digits in each PEM. The LSB in this string B of 256 digits (1024 bits) in PE_i indicates whether a_{1i} is zero or not; in general, the $j^{\underline{th}}$ bit in the string in PE_i refers to element a_{ji}. This method allows very efficient reconstitution of the actual rows but may still need too much storage space if the matrix is very large and very sparse. In this case, the following method is used instead.

2) The individual-tag method associates with each non-zero element of a matrix A a related positive integer t, called a tag. A tag matrix is constructed in which t_{ij} is zero if a_{ij} is zero and $t_{ij}=i$ if a_{ij} is non-zero. The tag matrix is then stored

with column j in PE_j and compacted in the same way used to compact A. Therefore, PEM_j will contain two strings of numbers: $a_1, a_2, \cdots, a_{n_j}$ and $t_1, t_2, \cdots, t_{n_j}$ where n_j is the number of non-zero elements of A in column j. a_i is the i^{th} non-zero element in column j of A; if this is element a_{kj}, then $t_i=k$. Each element t takes only three digits for matrices up to $4K \times 4K$. Note that n_j is normally different for each column of A but hopefully, if the zero elements of A are randomly distributed, no large variations exist between the number of non-zero elements in two columns.

The problem of multiplying two sparse matrices stored in either of the methods above is now considered. The broadcast method of multiplication (see Section 5.3) is used. Therefore, the only extra procedure needed is an efficient way to reconstruct the actual rows of the matrices. This is the purpose of the algorithms now described.

a) Expand in actual rows a sparse matrix stored according to the bit-matrix method. The rows must be expanded in order, from the first to the last. Fortunately, this is the order in which they are used in the broadcast method. Initially, the first digit b_1 of the bit string B is fetched from PEM in each PE and stored in sM(0). The address of the first element of each compacted column (i.e., the address of a_1) is sent via CAB to X_1. When each digit of the elements of the first row must be fetched, the PE's are enabled by the LSB of sM(0) during both the fetch and the subsequent increment of X_1 to point to the next digit. If the register to which the fetch is made is initially zeroed, the register will contain the correct row element after the fetch.

X_1 is also kept pointing to the appropriate element of the compact column since it is not advanced in PE_j when the actual row had a zero in column j. For the fetches of the three next rows, the three next bits of b_1 are used as enabling bits and then the next element b_2 of B is fetched, and so on. The extra time required to fetch the elements of B is probably easily overlapped with PE multiplications and the time to multiply two sparse matrices: $A \times B$ (each 1024×1024) stored according to the bit-matrix method is $D_A \times 43$ sec where D_A is the density of matrix A. Obviously, CU can analyze the broadcast elements and avoid broadcast of each zero element which decreases the multiplication time proportionately to the density of matrix A. It should be noticed that the optimum reduction factor is not simply D_A but $D_A \times D_B$. It is possible to devise an algorithm that achieves a reduction in time approaching the optimum value [13]; i.e., the algorithm also takes advantage of the sparseness of B to reduce multiplication time. However, the procedure is quite complex and will not be discussed here. It is also easy to see that the rows of the result can easily be compacted in the same bit-matrix representation if need be (i.e., if the product matrix is also sparse).

b) Expand in actual rows a sparse matrix stored according to the individual-tag method. As in case a, the rows must be expanded in order. In this case, however, the expansion procedure is less efficient. Initially, the first tag t_1 is fetched from PEM and compared for equality with the row number (i.e., one for the first row) sent via CDB. This fetch and comparison takes about 12 clocks since three digits must be compared and one of the operands is broadcast and does not have to be fetched. The result of the comparison, left in lcFF1, is then used to enable the fetch from PEM address X_1 and the subse-

quent increment of X_1. Therefore, an additional 1.2 μsec is needed to fetch each row in the individual-tag method. This cannot be overlapped with multiplications, as in case a because the arithmetic part of the PE must be used for the comparison.

6. CONCLUSIONS

The concept of an array computer with a very large number of relatively simple processing elements has been proven feasible; the PE hardware was described in great detail and the sections on operations and applications show that this hardware can be used quite efficiently. Obviously, several problems remain to be studied and the following considerations analyze these problems and offer some suggestions for further research.

Two areas are considered: 1) problems related with SPEAC in particular, and 2) problems related with the general architecture of array computers with many processing elements.

With respect to SPEAC in particular, the PE hardware has been painstakingly refined and optimized as far as one can get without an actual commitment to build the machine; a few questions remain to be answered and final "tuning" of the PE hardware must be performed, but these could be accomplished only with definite cost figures to analyze the cost-efficiency of different alternatives. Some of these alternatives were discussed in the section on implementation. A few specific points are:

a) The scratchpad memory sM introduced in the PE at a late stage in development has proven to be an impressive improvement, making possible a reduction by a factor of two to three in the times of floating-point operations. The study of applications also revealed that an increase in the capacity of sM will improve the performance in several areas. Therefore, the final size of sM must be carefully determined to optimize cost-efficiency. It is also interesting to notice that sM has performed so well

because of the relatively large values attributed to PEM access and cycle times (300 nsec and 500 nsec respectively). It now appears that these values are unduly pessimistic and depending on the final times obtained, the importance of sM will decrease and sM may be eliminated all together.

b) CU architecture was only sketched and a much more detailed design would be needed if the machine were to be built. Specifically the system of two queues for PE operation did not result in any substantial improvement for most operations. Since the system is quite expensive to implement and introduces serious complications in microprogramming, it should be dropped and only three queues used; one for I/O, one for PE, and one for CU instructions.

c) The possibility of overlapping PE instructions with I/O or CU instructions has proven very valuable in several applications. The system should be refined as suggested in Section 5.3-b to allow overlap not only in the use of PEM, but also in the use of sM.

d) Final minimizations in the number of connections and the number of chips per PE must be performed in view of the state of the art in integrated circuitry at the time of implementation. This field has advanced so rapidly that the picture has changed substantially within the last year. Specifically, one would need data about MOS - T^2L relative performance, equivalent gate densities obtainable per chip and cost of custom-built chips.

With respect to the field of array computers with a large number of

processing elements, the followings considerations are offered:

a) Software development for an array computer is a troublesome area as demonstrated by the arduous and sometimes frustrated efforts to develop a high-level language for ILLIAC IV. This was probably to be expected if one takes as a parallel the development of high-level software for sequential computers; it started only after a decade of painstaking machine-language programming. The lapse in the case of array computers should be much shorter since a whole body of knowledge about languages does exist and will be used as a basis. Nevertheless, array computer users seem to be condemned to a few years of assembly-language programming while software researchers gain the insight and experience needed to provide efficient and reliable high-level compilers.

It was expected at the beginning of this research that programming SPEAC would be one order of magnitude more difficult than programming ILLIAC IV just as programming IILIAC IV is one order of magnitude harder than programming conventional computers. Fortunately this has not been the case; programming SPEAC has been about as difficult as programming ILLIAC IV. Of course, this was mainly due to the fact that the size of the sample problems was selected to facilitate programming. The problem becomes more difficult when problems "smaller" than the size of the array must be handled efficiently and this is more and more frequent as the number of PE's increases.

If large array computers are to perform the role that is

expected of them, the user must be spared the task of knowing what each specific PE is doing, much in the same way as in conventional computers the user has been spared the task of keeping track of absolute memory addresses. An initial step in this direction is provided by N. R. Lincoln. In a recent paper [10], he proposes a radically new technique for using array computers in such problems as compiling, which have so far been considered typically non-parallel (that is, unsuitable for these machines). Such techniques, if successful, could increase tremendously the area of application of SPEAC. The study of the performance of SPEAC in pattern matching problems, which was discussed in Section 5.4, has shown that it can perform very efficiently the basic tasks required in Lincoln's scheme.

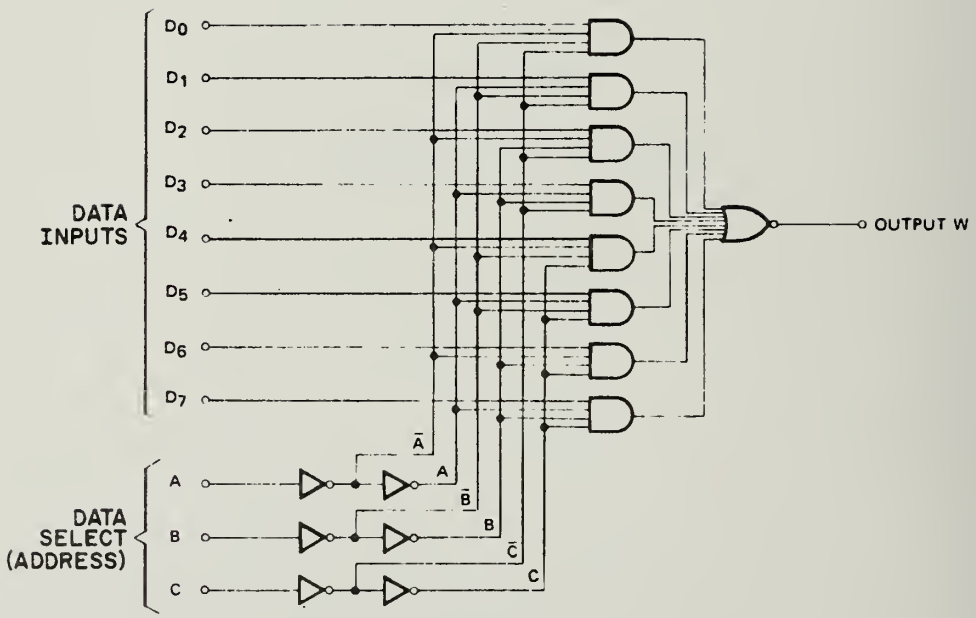
b) One very promising idea has been recently proposed to help solve the problem of handling efficiently problems "smaller" than the size of the array in computers of the type of SPEAC. It consists of linking groups of PE's together in a hardware-implemented fashion so that a group of PE's would be able to function as a single PE with speed roughly proportional to the number of actual PE's in the group. The problem is reasonably complex and will require considerable research but the possibilities are far-reaching; this method would not only make it much easier to use efficiently computers of the scale of SPEAC, but it would also make practical array computers with tens and even hundreds of thousands of very simple PE's.

c) Finally, one very long-range research project would be to inves-

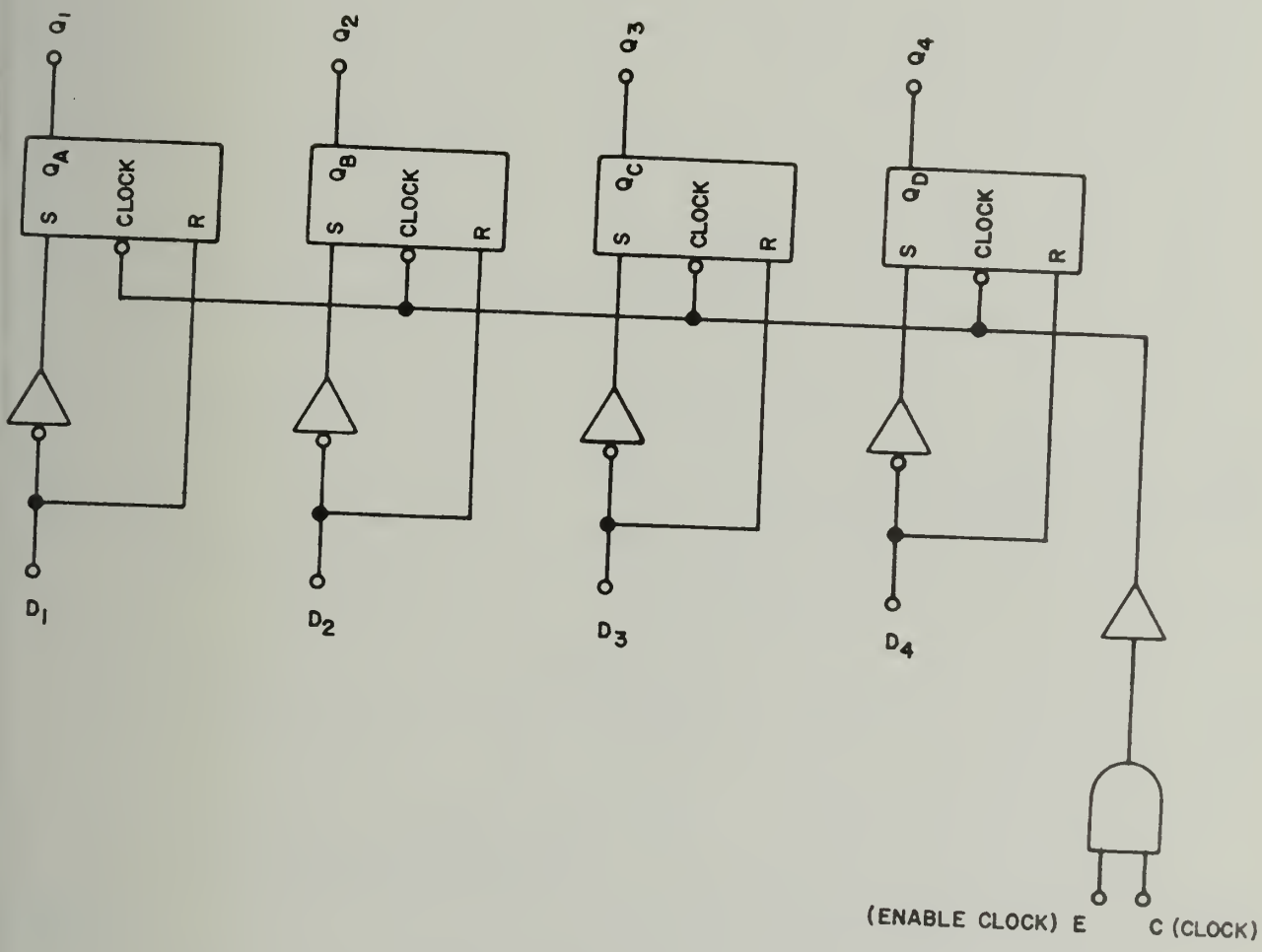
tigate how far one could go with the number of elements in a parallel processor. The approach described above allows one to envision a processing unit composed of many similar "PE's" linked together in a fail-soft configuration, much like the individual cells in a brain. If one PE fails, the only immediate effect would be a slight reduction in the speed of the processor as a whole.

APPENDIX A

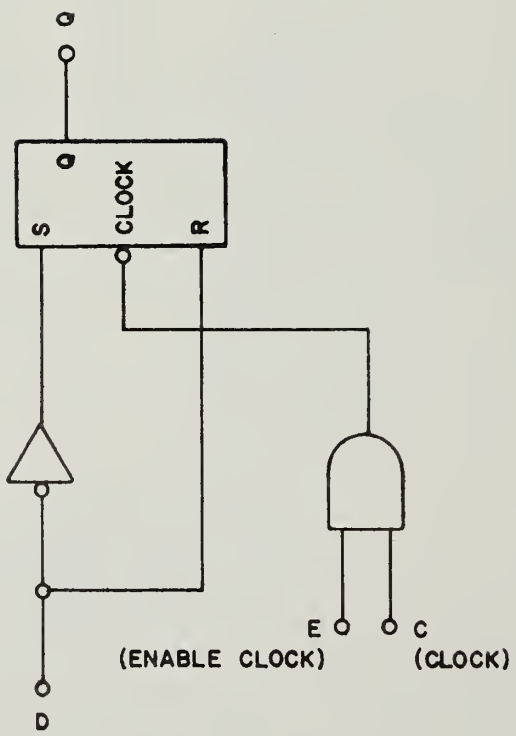
PACKAGE LOGICAL DIAGRAMS



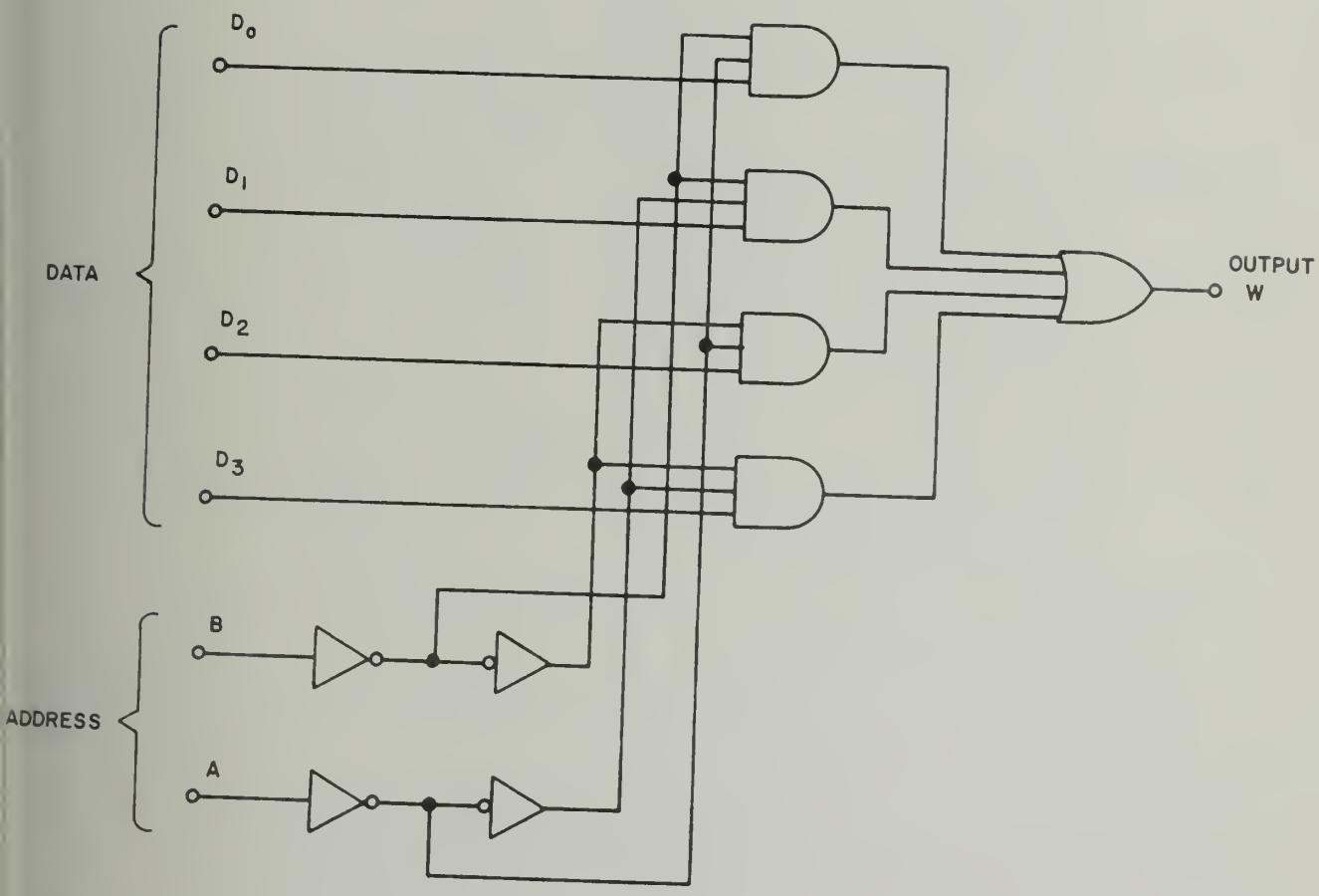
Package 1. One-out-of-eight Selector without Strobe



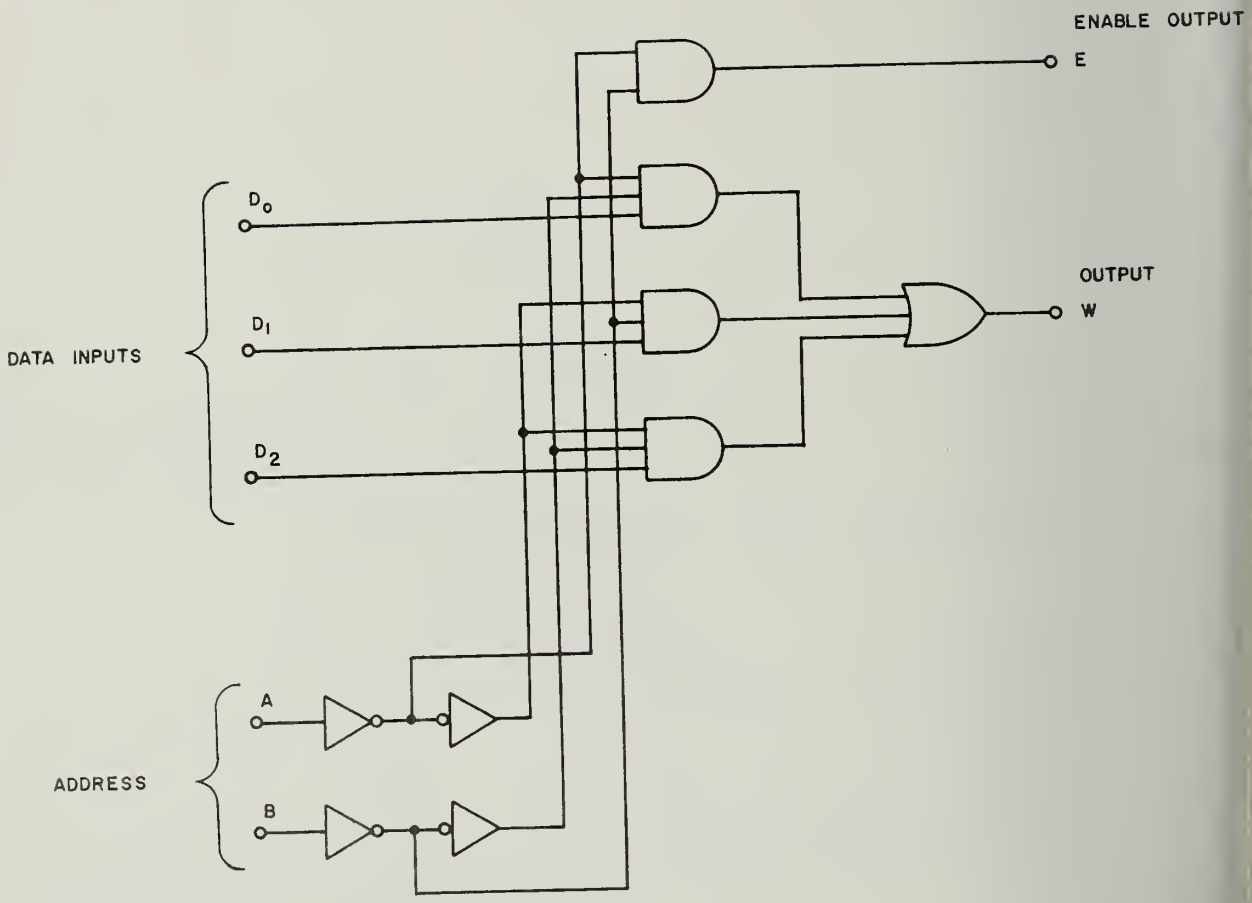
Package 2. Quad Type D Flip-flop with Enable on the Clock



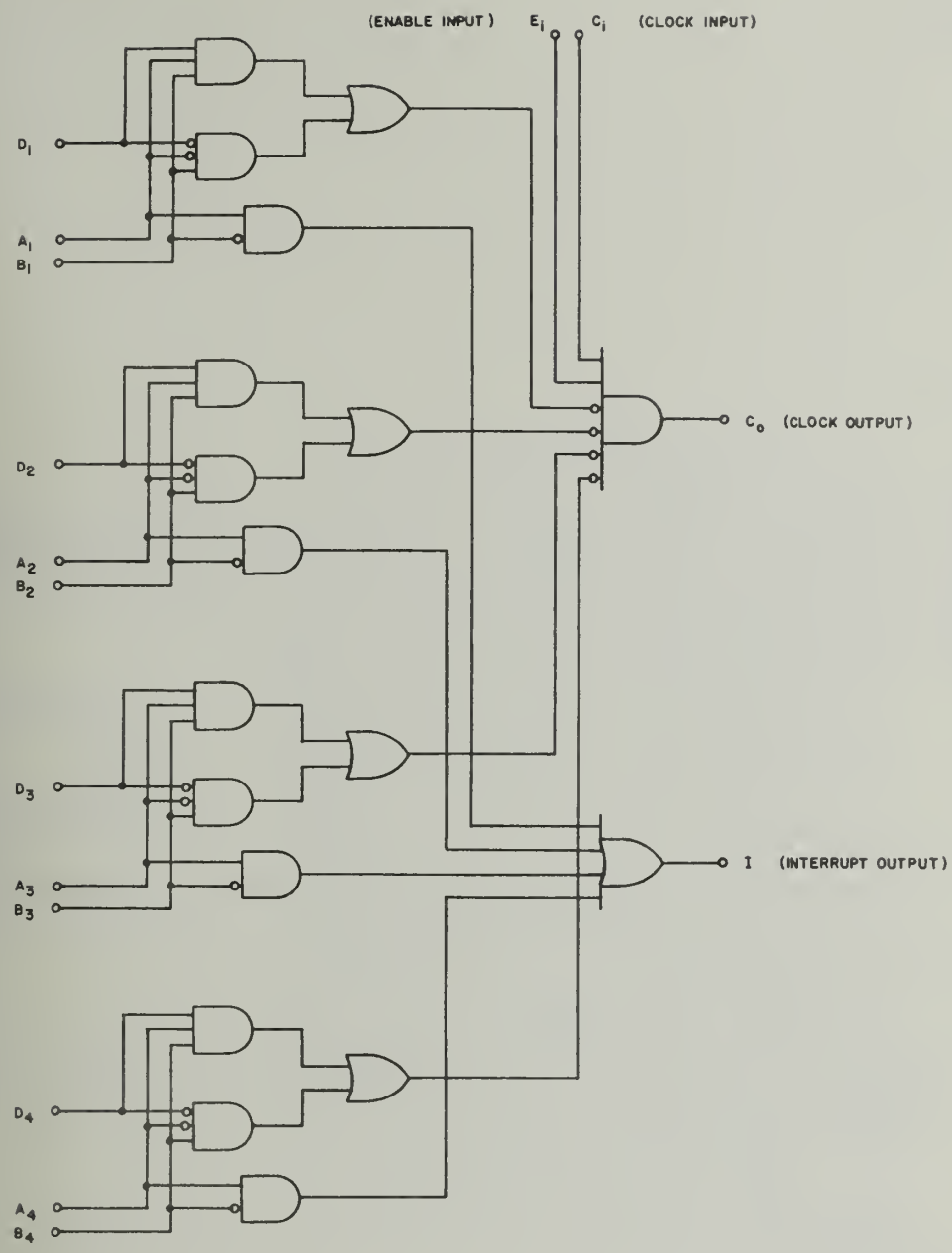
Package 3. Type D Flip-flop with Enable on the Clock



Package 4. One-out-of-four Selector without Strobe



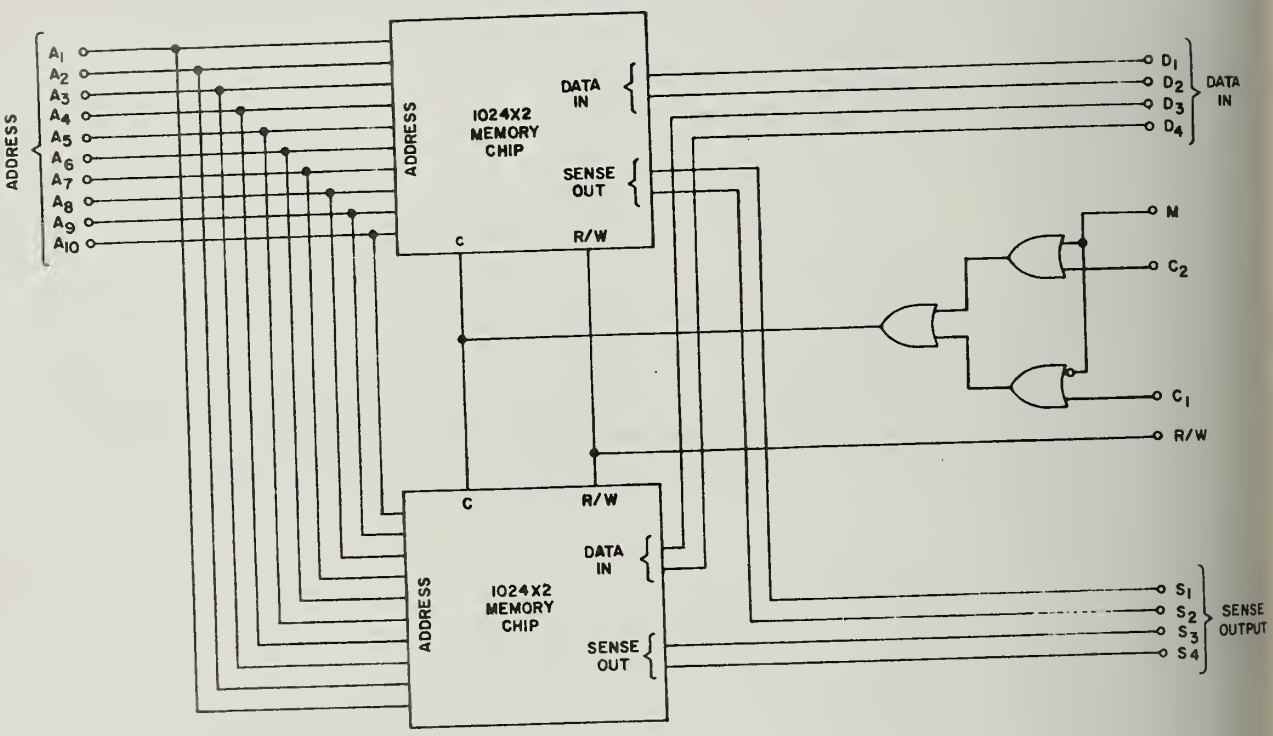
Package 5. One-out-of-three Selector with Enable Decoding



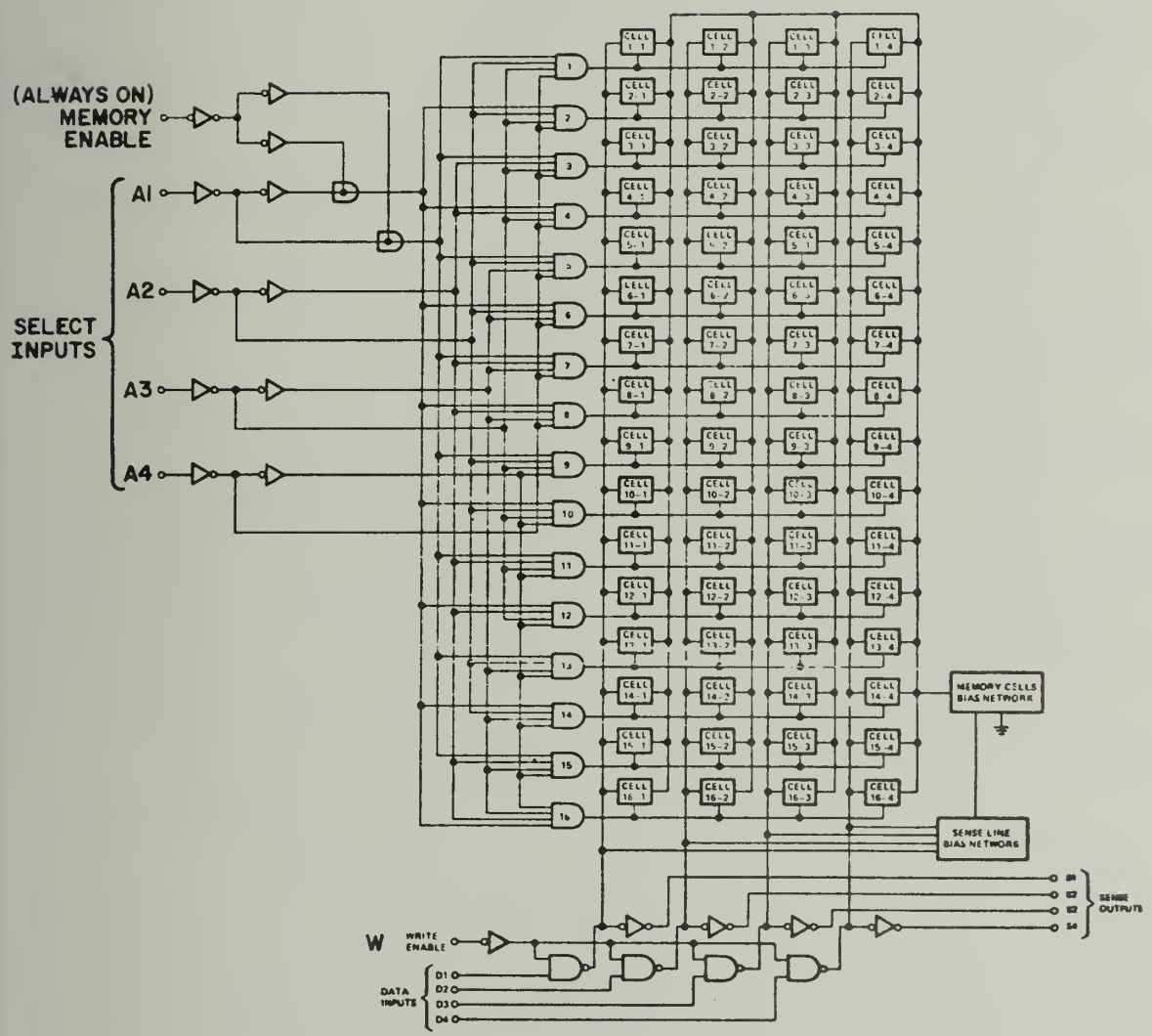
Note: The Function is as follows for each lcFF:

A B	Function
0 0	Do nothing, i.e., the lcFF is not used
1 0	Use the lcFF to control the interrupt wire; the interrupt wire will assume the logical level of the lcFF
0 1	Enable the PE (i.e., allow the clock to reach the registers) when the lcFF contains a ZERO
1 1	Enable the PE when the lcFF contains a ONE

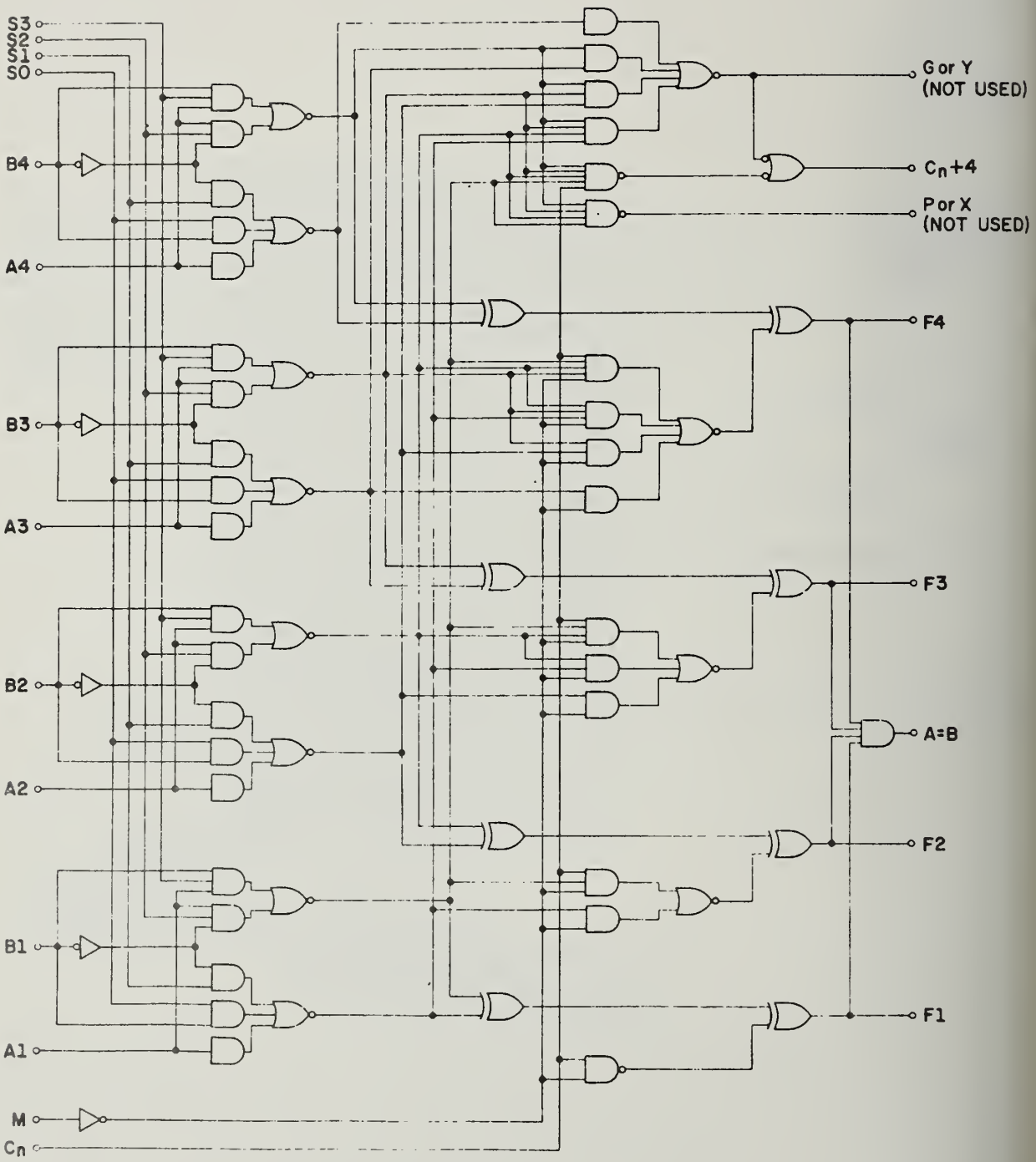
Package 6. Enable and Interrupt Control



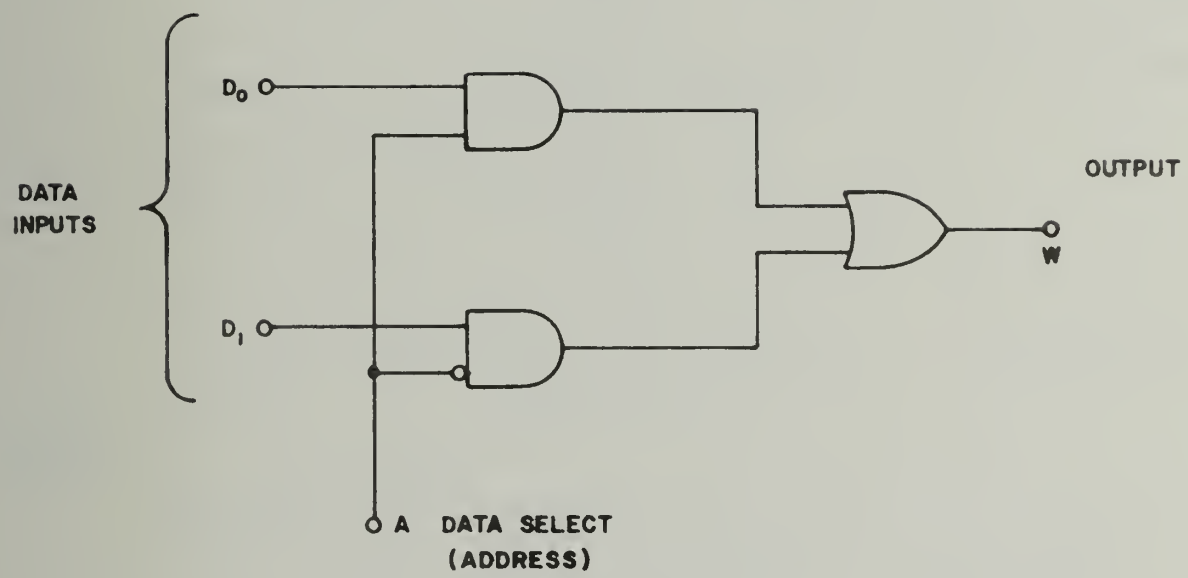
Package 7. PEM - 1 Module



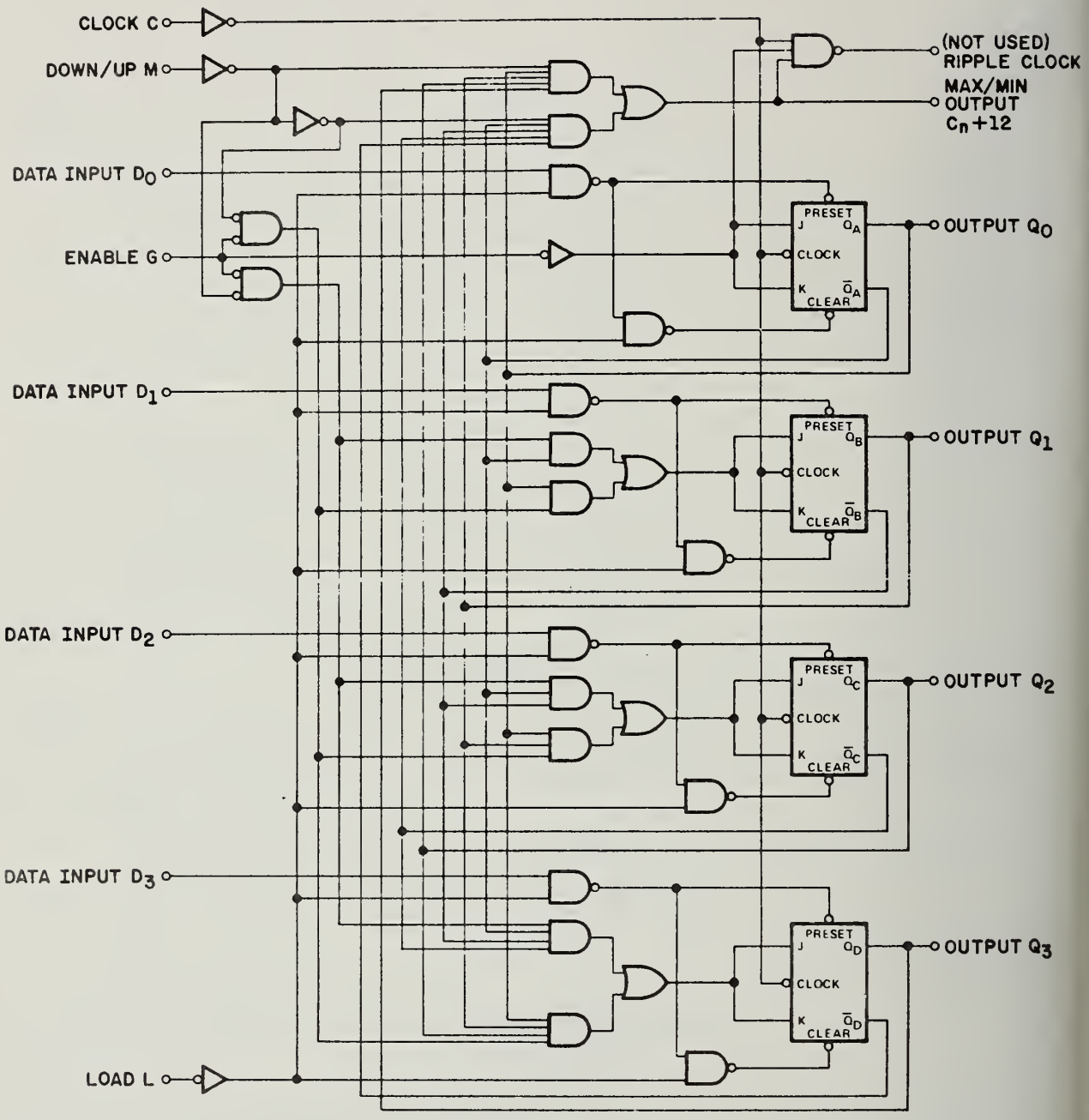
Package 8. 64-bit Scratchpad Memory
(16 4-bit Words)



Package 9. Arithmetic/Logic Unit

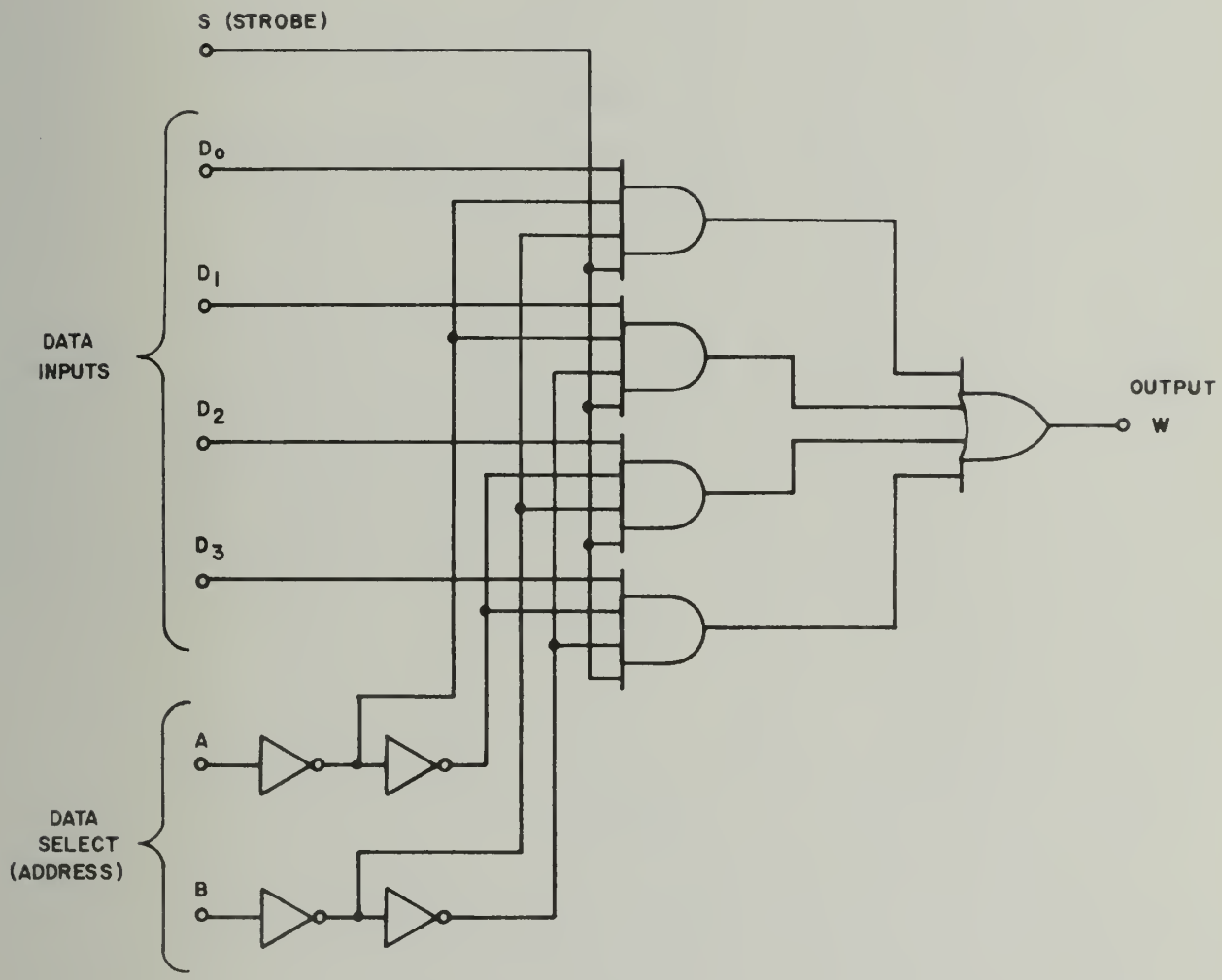


Package 10. One-out-of-two Selector without Strobe

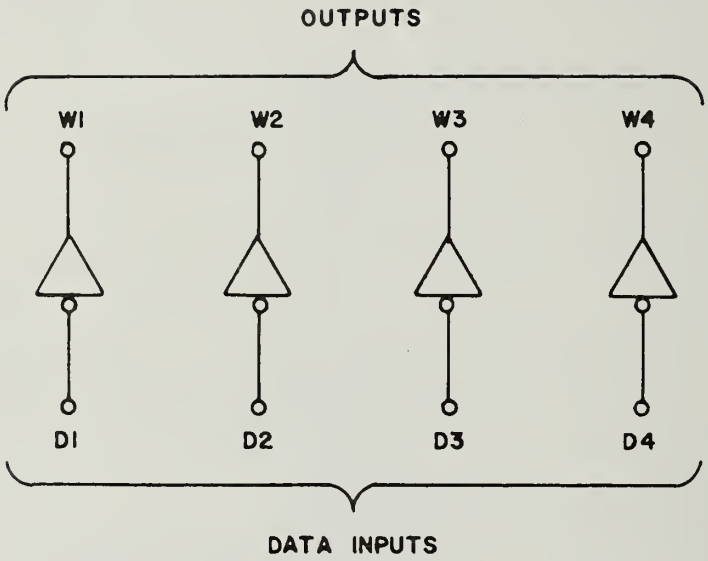


Note: When cascading, G input goes to least significant hexadecimal digit and C_{n+12} output comes only from most significant hexadecimal digit; G_{i+1} is connected to $C_{(n+12)i}$, for all not externally connected G and C_{n+12}.

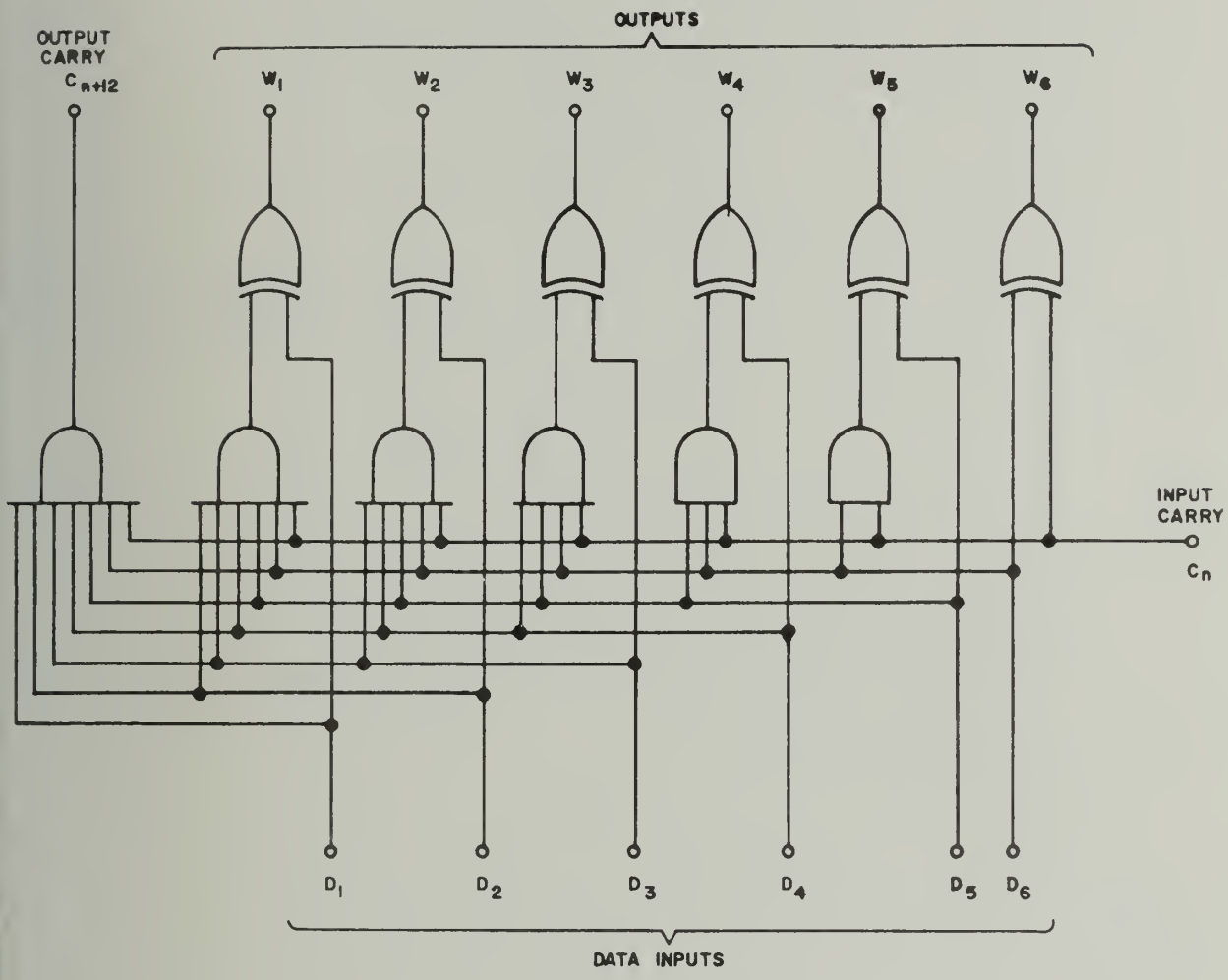
Package 11. 4-bit Up/Down Counter, Parallel In/Out



Package 12. One-out-of-four Selector with Strobe

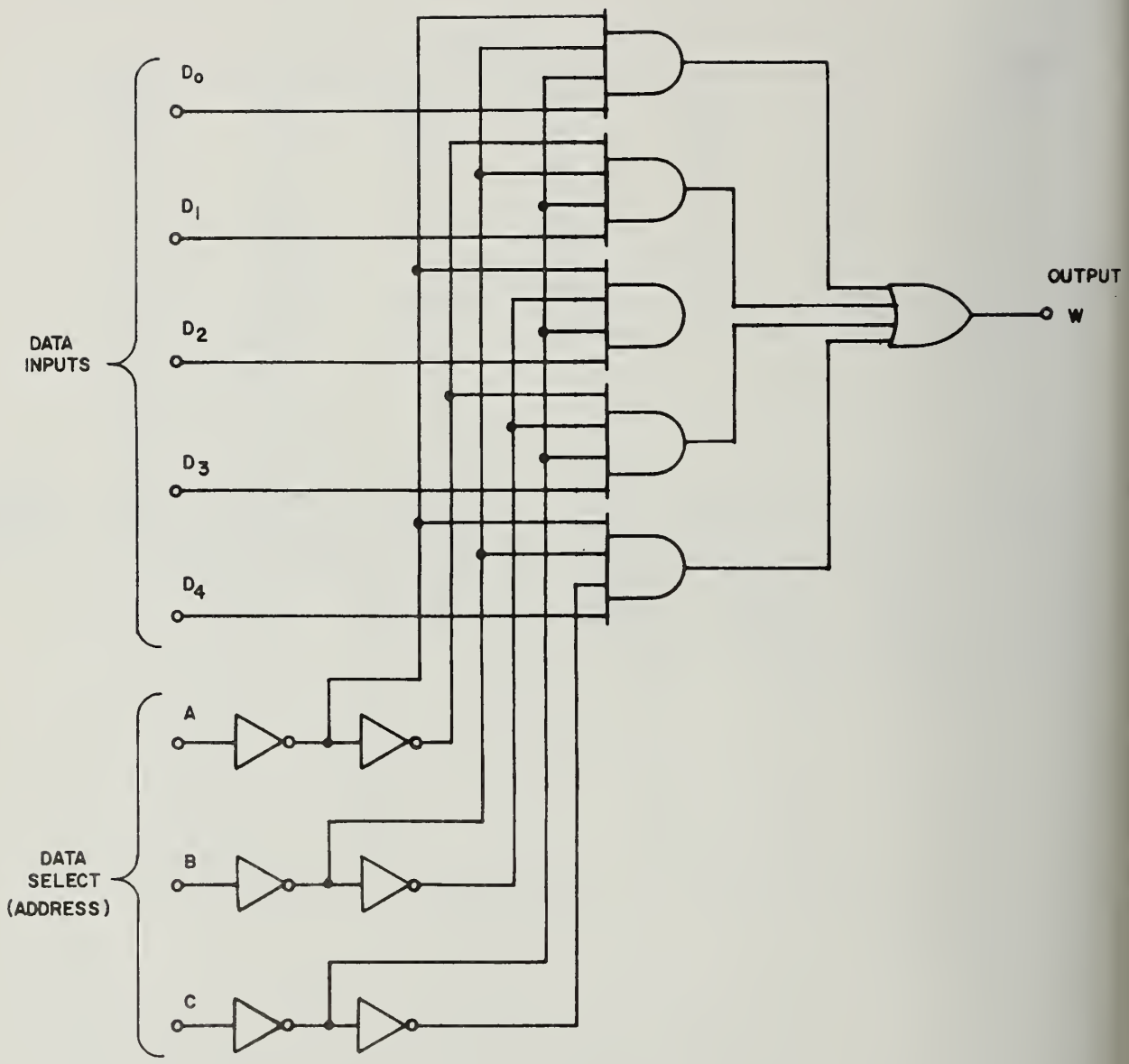


Package 13. Quad Inverter



Note: When cascading, C_{n+12} output comes only from most significant package; C_n input to the least significant package is "1." Input $(C_n)_{i+1}$ is connected to output $(C_{n+12})_i$

Package 14. Increment-by-one Network (16 bits)



Package 15. One-out-of-five Selector without Strobe

APPENDIX B

MICROSEQUENCE FOR 32-BIT FLOATING-POINT MULTIPLICATION

This is a detailed listing of the microsequences sent by CU to each PE to perform the multiplication: $a \times b = c$, where each number is in the following format:

$$a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$$

a_0 is the mantissa LSD (least significant digit)

a_5 is the mantissa MSD (most significant digit)

a_{60} (i.e., the low order bit of a_6) is the mantissa sign bit

a_7, a_{63}, a_{62} and a_{61} constitute the exponent; a_{61} is the low order bit of the exponent.

The exponent is in excess notation and the mantissa in sign and magnitude. The exponent base is 16. a_0 is the low address in the PEM.

The following abbreviations are used in the microsequences:

A ← B which means that register A is loaded with the contents of register B.

sM(X) or PEM(X) which means the contents of the location with address X in sM or PEM; X can be a literal or a register in which case the contents of the register are taken as the address. When X is a literal, it is sent via CAB.

CAB(a) or CDB(a) which means that data a is sent via the common bus.

En(i,ON) or En(i,OFF) which means that the enable function is attributed to lcFFi ON or OFF.

Each microsequence is numbered with two PE clock counts: maximum and minimum. The minimum count assumes that the two buses are available and maximum overlap can be achieved; the maximum count assumes that only one bus is available at all times for PE operation. CAB and CDB are assumed always available.

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
1	1	$X_1 \leftarrow$ CAB(address (a_0))	Address registers are loaded with mantissas' LSD addresses
2	2	$X_2 \leftarrow$ CAB(address (b_0))	
3	3	$A_r \leftarrow$ PEM(X_1); sM(0) $\leftarrow$ PEM(X_1)	PEM read; takes 3 clocks
6	3	B $\leftarrow$ PEM(X_2)	If overlap is possible, one extra clock is needed to store in sM
8	6	sM(6) $\leftarrow$ PEM(X_2)	
9	6	$A_m \leftarrow$ CDB(0); $A_c \leftarrow$ CAB(0); Incr X_1; Incr X_2	Ready to start multiplication; X_1, X_2 are ready to access the next digits
10	7	MF(1, X_1, *, 1, *)	See note a for the meaning of MF; m_0 is completed
15	12	MF(7, X_2, 7, 0, S)	
20	17	MF(2, X_1, 6, 2, *)	m_1 is completed
25	22	MF(*, *, 7, 1, S)	
30	27	MF(8, X_2, 8, 0, S)	
35	32	MF(3, X_1, 6, 3, *)	m_2 is completed
40	37	MF(*, *, 7, 2, S)	
45	42	MF(*, *, 8, 1, S)	
50	47	MF(9, X_2, 9, 0, S)	
55	52	MF(4, X_1, 6, 4, *)	m_3 is completed
60	57	MF(*, *, 7, 3, S)	
65	62	MF(*, *, 8, 2, S)	
70	67	MF(*, *, 9, 1, S)	
75	72	MF(10, X_2, 10, 0, S)	

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
80	75	MF(5, X_1, 6, 5, *)	m_4 is completed
85	82	MF(*, *, 7, 4, S)	
90	87	MF(*, *, 8, 3, S)	
95	92	MF(*, *, 9, 2, S)	
100	97	MF(*, *, 10, 1, S)	
105	102	MF(11, X_2, 11, 0, S)	
110	107	ML(7, 5, 0)	See note b for the meaning of ML, m_5 is loaded in sM(0)
116	113	MF(7, X_1, 8, 4, S)	a_6 is loaded in sM(7) for future use
121	118	MF(*, *, 9, 3, S)	
126	123	MF(*, *, 10, 2, S)	
131	128	MF(*, *, 11, 1, S)	
136	133	ML(8, 5, 1)	m_6 is loaded in sM(1)
142	139	MF(8, X_1, 9, 4, S)	a_7 is loaded in sM(8) for future use
147	144	MF(*, *, 10, 3, S)	
152	149	MF(*, *, 11, 2, S)	
157	154	ML(9, 5, 2)	m_7 is loaded in sM(2)
163	160	MF(9, X_2, 10, 4, S)	b_6 is loaded in sM(9) for future use
168	165	MF(*, *, 11, 3, S)	
173	170	ML(10, 5, 3)	m_8 is loaded in sM(3)
179	176	MF(10, X_2, 11, 4, S)	b_7 is loaded in sM(10) for future use

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
184	181	ML(11, 5, 4)	m_9 is loaded in sM(4)
190	187	ML(*, *, 5)	m_{10} is loaded in sM(5)
191	188	$X_1 \leftarrow$ CAB(address (c_0))	
192	189	$X_2 \leftarrow$ CAB(0)	X_1 and X_2 are prepared to write the result
196	193	lcFF1 $\leftarrow$ (A_m=CDB(0)); sM(6) $\leftarrow A_m$	m_{11} is loaded in sM(6)
197	193	$A_m \leftarrow$ CDB(0)	
198	194	En(1,ON); $A_m \leftarrow$ CDB(0010); Incr X_2	See note c
199	195	ST	c_0 is stored in PEM; see note d for the meaning of ST
205	201	ST	c_1 is stored in PEM
211	207	ST	c_2 is stored in PEM
217	213	ST	c_3 is stored in PEM
223	219	ST	c_4 is stored in PEM
229	225	ST	c_5 is stored in PEM
235	231	ST; wait on Event #1	c_6 is stored in PEM; see note e
241	237	ST; wait on Event #2	c_7 is stored in PEM
			Exponent computation starts now
00	200	B $\leftarrow$ sM(7); $A_c \leftarrow$ CAB(0)	B is loaded with LSD of exponent of a
06	201	$A_m \leftarrow$ (B-A_m); C_n=1; lcFF4 $\leftarrow C_{n+4}$	A_m is still as in note c
12	206	Shift A_r, A_m right 4; B $\leftarrow$ sM(8)	B is loaded with MSD of exponent of a
18	207	$A_m \leftarrow$ (B-A_m); C_n=lcFF4	

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
224	212	Shift A left 4; B ← sM(9)	Now have in A_c, A_m: exp(a)-1 if m_{11}=0 and exp(a) if $m_{11}\neq 0$
230	213	$A_r \leftarrow (A_m \oplus B)$	A_{r_0} now contains the sign of c
236	214	$A_m \leftarrow (A_m$ AND CDB(1110))	Set sign bit to zero in A_m
242	215	$A_m \leftarrow (A_m+B)$; C_n=0; lcFF4 ← C_{n+4}	
243	216	$A_{m_0} \leftarrow A_r$	A_m now contains LSD of exp(c)
244	218	sM(7) ← A_m; cause Event #1	
245	224	Shift A_m right 4; B ← sM(10)	A_m has MSD of exp(a) and B has MSD of exp(b)
246	225	$A_m \leftarrow (A_m+B)$; C_n=lcFF4; lcFF4 ← C_{n+4}	
247	226	$A_m \leftarrow (A_{m_1} \oplus$CDB(1000)); shift A_r right 4	Correct sum in excess notation by complementing MSB
248	230	sM(8) ← A_m; cause Event #2; shift A_m left 4	
249	231	A_{m_2}, A_{m_1}, A_{m_0} ← LC	Start detection of exponent overflow or underflow; see note f
250	232	lcFF1 ← (A_m=LC)	
251	233	Interrupt on lcFF1 ON	
251	241		End of the operation

Notes:

a) MF(a, b, c, d, S) is defined as the following set of five microsequences:

1) Add and shift; $\underbrace{sM(a) \leftarrow PEM(b)}_{I}$

2) Add and shift

3) Add and shift $\overbrace{\text{Incr (b)}}^{II}$; $\overbrace{B \leftarrow sM(c)}^{III}$

4) Add and shift;

5) $A_r \leftarrow sM(d)$; shift A_c, A_m left 4 (IV)

If a and b are *'s then portions I and II are absent; if c is a * then portion II is absent; if S is replaced by a * then portion IV is absent. MF can perform the following: a) multiply two digits, b) fetch from PEM and store in sM a digit to be used in the next multiplication, and c) load A_m and B with the two digits needed in the next multiplication.

b) ML(a, b, c) is defined as the following set of six microsequences:

1) Add and shift

2) Add and shift

3) Add and shift

4) Add and shift; $\underbrace{B \leftarrow sM(a)}_{I}$

5) $sM(c) \leftarrow A_r$

6) $\underbrace{A_r \leftarrow sM(b)}_{II}$

If a is a * then portion I is absent; if b is a * then portion II is absent. ML multiplies two digits, stores the MSD of the product in sM and loads A_m and B with the two digits needed in the next multiplication.

c) At this stage, X_2 points to m_5 (in sM(0)) if $m_{11}=0$ and to m_6 (in sM(1)) if $m_{11}\neq 0$. Therefore, X_2 points to c_0. Also, A_m contains 0000 if $m_{11}\neq 0$ and 0010 if $m_{11}=0$ to prepare for the correction in the exponent.

d) ST is defined as the following set of six microsequences:

1) PEM(X_1) ← sM(X_2)
2) Wait for writing in PEM
3) Wait for writing in PEM
4) Wait for writing in PEM
5) Wait for writing in PEM
6) Incr X_1, Incr X_2

ST stores the digits of the product in PEM. This is overlapped as much as possible with the computation of the exponent.

e) The wait in this microsequence assures that the exponent will be written in PEM only after it is computed.

f) In excess notation addition, there is an overflow if the carry from the MSB is equal to the MSB of the sum before the necessary correction which consists of complementing the MSB.

APPENDIX C

MICROSEQUENCE FOR 32-BIT FLOATING-POINT ADDITION

This is a detailed listing of the microsequences sent by CU to each PE to perform the addition: a + b = c. Number format, notation and abbreviations used are as listed in the introduction to Appendix B.

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
1	1	$X_1 \leftarrow$ CAB(address (a_7))	Address registers are loaded with address of MSD of the exponents
2	2	$X_2 \leftarrow$ CAB(address (b_7))	
3	3	$A_m \leftarrow$ PEM(X_1); sM(3) $\leftarrow$ PEM(X_1)	PEM read; takes 3 clocks
6	3	B $\leftarrow$ PEM(X_2)	If overlap is possible, one extra clock is needed to store in sM
8	6	sM(1) $\leftarrow$ PEM(X_2)	
9	7	lcFF1 $\leftarrow$ (A_m=B); lcFF4 $\leftarrow$ (A_m< B); Shift A_c left 4, Decr X_1, Decr X_2	Comparison of exponents starts
10	8	$A_m \leftarrow$ PEM(X_1); sM(2) $\leftarrow$ PEM(X_1)	Read the LSD's of the exponents
13	8	B $\leftarrow$ PEM(X_2)	
15	11	sM(0) $\leftarrow$ PEM(X_2)	
16	12	En(1,ON); lcFF4 $\leftarrow$ (A_m< B); Shift A_c left 4	lcFF4 is now ON iff exp(a)$\geq$ exp(b); A_c contains exp(a)
17	13	$A_m \leftarrow$ (A_m AND CDB(0001))	All bits except sign are zeroed
18	14	Shift A_r right 4; $A_m \leftarrow$ B	
19	15	$A_m \leftarrow$ (A_m AND CDB(0001))	All bits except sign are zeroed
20	16	lcFF1 $\leftarrow$ (A_m=A_r)	lcFF1 is now ON if sign(a)=sign(b)
21	17	En(4,ON); $A_r \leftarrow$ sM(0); $A_c \leftarrow X_1$	Interchange exponents and addresses in PE's in which exp(a)$\geq$ exp(b)
22	18	En(4,ON); $A_m \leftarrow$ sM(1); $X_1 \leftarrow X_2$	
23	19	En(4,ON); B $\leftarrow$ sM(2); $X_2 \leftarrow A_c$	
24	20	En(4,ON); sM(0) $\leftarrow$ B; shift A left 4	
25	21	En(4,ON); B $\leftarrow$ sM(3); shift A left 4	

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
26	22	En(4,ON); sM(1) ← B	
27	23	Shift A right 4; sM(2) ← LC	See note a
28	24	B ← sM(0)	Exponent subtraction now starts
29	24	A_m ← (A_m OR CDB(0001))	Sets sign bit to one so that it does not interfere with subtraction
30	25	A_r ← (B-A_m); C_n=1; lcFF4 ← C_{n+4}; B ← sM(1); shift A_m right 4	
31	26	A_m ← (B-A_m); C_n=lcFF4; A_c ← CAB(0)	
10	8	Decr X_1, Decr X_2	These six clocks are overlapped with previous ones; they make X_1 point to a_0 and X_2 point to b_0
15	11	Decr X_1, Decr X_2	
16	12	Decr X_1, Decr X_2	
17	13	Decr X_1, Decr X_2	
18	14	Decr X_1, Decr X_2	
19	15	Decr X_1, Decr X_2	
32	27	Shift A right 1; A_c ← X_2	See note b
33	28	lcFF1 ← (A_m=CDB(0)); B ← A_r	
34	29	lcFF2 ← CDB(0); A_r ← CDB(0)	
35	30	En(1,OFF); lcFF2 ← CDB(0010); shift A right 4	Ready now to perform mantissa alignment; see note c
36	31	A_m ← (A_m+B); C_n=0; lcFF4 ← C_{n+4}	
37	32	En(4,ON); Incr A_c	
38	33	Shift A left 4	
39	34	X_2 ← A_c	Mantissa alignment completed
40	34	A_c ← CAB(FFF-N_m+1)	Prepare trap in A_c; see note d

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
41	35	Shift A right 4; $A_c \leftarrow$ CAB(FFF)	
42	36	$A_m \leftarrow (A_m+B)$; $C_n=0$; lcFF4 $\leftarrow C_{n+4}$; B $\leftarrow$ PEM(X_2)	B still had the difference of the exps; it is reloaded with the first operand
43	37	En(4,ON); En(2,OFF); Incr A_c; lcFF2 $\leftarrow C_{n+12}$	
44	37	lcFF1 $\leftarrow$ sM(2); shift A_c left 4	Trap is completed; lcFF1 is ON only if signs are equal
45	38	$A_m \leftarrow$ PEM(X_1)	Fetch the second operand
48	41	ADFI(4)	The actual addition starts now; see note e for the meaning of ADFI, ADF and $\overline{\text{AD}}$
57	47	ADF(5)	
66	53	ADF(6)	
75	59	ADF(7)	
84	65	ADF(8)	
93	71	AD(9)	Addition completed; now find out sign of result and if recomplementation is needed; see note f.
96	74	$A_m \leftarrow$ LC; B $\leftarrow$ LC	
97	75	sM(3) $\leftarrow A_m$; shift A right 4	
98	76	Shift A left 1	
99	77	$A_m \leftarrow (\overline{A_m}$ AND B)	
100	78	lcFF1 $\leftarrow A_m$	lcFF1 is ON if recomplementation is needed
101	78	B $\leftarrow$ CDB(0)	
102	79	$A_m \leftarrow$ sM(4)	Ready to start recomplementation
103	80	RCI(5,4)	See note g for meaning of RC and RCI
105	82	RC(6,5)	

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
107	84	RC(7,6)	
109	86	RC(8,7)	
111	88	RC(9,8)	
113	90	RC(*,9)	Recomplementation completed
115	92	$A_m \leftarrow sM(0)$	Now set up sign of result; i.e., change the sign of the exponent in sM(0), sM(1) if recomplementation was needed.
116	93	En(1,ON); $A_m \leftarrow (A_m \oplus CDB(0001))$	
117	94	$sM(0) \leftarrow A_m$	
118	95	$A_m \leftarrow sM(3)$; $B \leftarrow sM(3)$	sM(3) contains MSB ON if there was a final output carry and LSB ON if sign(a)=sign(b)
119	96	Shift A_c left 4; $X_1 \leftarrow CAB(FFF)$	
120	97	Shift A_m right 1; lcFF1 $\leftarrow$ CDB(0001)	
121	98	$A_m \leftarrow (A_m$ AND B)	
122	99	lcFF4 $\leftarrow A_m$	lcFF4 is now ON if there was an "overflow."
123	99	$A_m \leftarrow sM(1)$; $A_c \leftarrow CAB(0)$	
124	100	Shift A_c left 4; $A_m \leftarrow sM(0)$	
125	101	Shift A_c left 4; $A_r \leftarrow CDB(0)$	
126	102	Shift A_c right 1; sM(10) $\leftarrow$ CDB(0001)	
127	103	$X_2 \leftarrow A_c$; $A_m \leftarrow sM(9)$	X_2 now contains exp(a) without the sign
128	104	CZ(8)	See note h for the meaning of CZ

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
130	106	CZ(7)	
132	108	CZ(6)	
134	110	CZ(5)	
136	112	CZ(4)	
138	114	CZ(*)	
140	116	En(4,ON); Incr X_2	This adds 1 to the exp if there was "overflow"
141	117	$A_c \leftarrow X_2$; $A_r \leftarrow$ CDB(0); $A_m \leftarrow$ CDB(0)	
142	118	Shift A right 4	
143	119	Shift A left 1	
144	120	$A_{m_0} \leftarrow$ sM(0)	Insert the sign back in the exponent
145	121	sM(0) $\leftarrow A_m$	
146	122	Shift A right 4	
147	123	sM(1) $\leftarrow A_m$; shift A_m right 4; B $\leftarrow$ CDB(0)	Final exponent is now in sM(0), sM(1); prepare to detect exponent overflow or underflow
148	124	lcFF1 $\leftarrow (A_m=B)$; $A_c \leftarrow X_1$	
149	125	Interrupt on lcFF1 OFF; $X_2 \leftarrow$ CAB(4)	lcFF1 OFF means exponent overflow or underflow
150	126	En(4,ON); $X_2 \leftarrow$ CAB(5)	
151	127	Incr A_c; lcFF2 $\leftarrow C_{n+12}$; $X_1 \leftarrow$ CAB(address(c_0)); B $\leftarrow$ CDB(0)	Ready to start storing the result
152	128	WR	See note i for the meaning of WR
160	136	WR	

Maximum PE Clock	Minimum PE Clock	Microsequence	Comments
168	144	WR	
176	152	WR	
184	160	WR	
192	168	WR	Mantissa is stored in PEM
200	176	$PEM(X_1) \leftarrow sM(0)$	Now store exponent in PEM
205	181	Incr X_1	
206	182	$PEM(X_1) \leftarrow sM(1)$	
214	190		End of the operation

Notes:

a) At this stage, the situation is as follows: lcFF1 is ON if the signs are equal, OFF otherwise; A_c, A_m contains the smaller exponent; the larger exponent is stored in sM(0), sM(1); in sM(2) the LSB is a one if the signs are equal and a zero otherwise.

b) At this point the situation is that A_m, A_r contains the difference of the exponents. If A_m is non-zero, then b will not participate in the sum (since the exponent difference is too large) and lcFF2 is set ON in PE's in which this happens.

c) Mantissa alignment is performed by adding the exponent difference (which is in B) to the address of b which is in A_c, A_m. The modified address of b is then returned to X_2.

d) A_c will be used as a counter which yields an overflow when all digits of b have been used. For PE's in which this overflow (which is stored as a lcFF2 ON) has appeared, digits of b are replaced by zeros before the addition.

e) ADF(a) (add and fetch) is defined as the following set of microsequences:

1,1 - En(2,ON); B ← CDB(0); Incr X_1; Incr X_2

2,2 - En(2,OFF); Incr A_c; lcFF2 ← C_{n+12}

3,3 - $A_r \leftarrow (A_m \pm B)$; C_n=lcFF4; lcFF4 ← C_{n+4}; $A_m \leftarrow PEM(X_1)$; lcFF1 OFF causes subtraction instead of addition

6,3 - $B \leftarrow PEM(X_2)$

9,6 - $sM(a) \leftarrow A_r$

ADF takes a minimum of six clocks and the normal time is nine clocks.

ADFI is similar to ADF but in clock (2,2) C_n is set to $\overline{\text{lcFF1}}$ instead of to lcFF4. ADFI is used for the first addition and takes as long as ADF.

AD is similar to ADF but no new fetch is performed. It is used for the last addition and takes only three clocks.

f) The rules are: for a±b=c, sign(c)=sign(a) and no recomplementation is needed unless sign(a)≠sign(b) AND lcFF4 is OFF at the end of the operation. In this case, sign(result)=$\overline{\text{sign(a)}}$=sign(b) and recomplementation must be performed. An overflow occurs when sign(a)=sign(b) AND lcFF4 is ON at the end of the operation.

g) RC(a,b) (recomplement) is defined as the two following microsequences:

1) $A_r \leftarrow ((\overline{A_m} \vee B)+B+1)$; $A_m \leftarrow sM(a)$; C_n=lcFF4; lcFF4 $\leftarrow C_{n+4}$

2) En(1,ON); $sM(b) \leftarrow A_r$

- If a is a *, then A_m is not loaded on the first microsequence.
- The arithmetic function above performs recomplementation when B=0.
- RCI(a,b) is used for the recomplementation of the first digit; it is similar to RC but in the first microsequence C_n=1 instead of C_n=lcFF4.

h) CZ(a) (count zeros) is defined as the following set of two microsequences:

1) En(1,ON); En(4,OFF); lcFF1 $\leftarrow (A_m=B)$; $A_m \leftarrow sM(a)$

2) En(1,ON); En(4,OFF); Decr X_1; Decr X_2

If a is a * then A_m is not reloaded in the first microsequence. This function decrements X_1 and X_2 if A_m is zero (and has always been zero previously) and if there was no "overflow" which is signaled by lcFF4 OFF. Since X_1 contains initially all 1's, a trap is formed to yield a carry when the number of leading zeros is added to it. Since X_2 contains initially the larger exponent, the exponent of the result is formed by subtracting one out of X_2 for each leading zero.

i) WR (write) stands for the following set of eight microsequences:

1) En(2,ON); B ← sM(X_2); Incr X_2

2) En(2,OFF); Incr A_c; lcFF2 ← C_{n+12}

3) PEM(X_1) ← B

4) Wait for writing in PEM

5) Wait for writing in PEM

6) Wait for writing in PEM

7) Wait for writing in PEM

8) Incr X_1

WR stores the sum of the mantissas in PEM and also takes care of eliminating leading zeros. The trap in A_c signals when all leading zeros (which are transformed in trailing zeros) are eliminated.

LIST OF REFERENCES

[1] Control Data Corporation. "The STAR Computing System." A technical proposal to The Atomic Energy Commission. December 1966.

[2] Slotnick, D. L., et. al. "The ILLIAC IV Computer," IEEE Transactions on Computers. Volume C-17, Number 8 (August 1968), pp. 746-757.

[3] Kuck, D. J. "ILLIAC IV Software and Application Programming," IEEE Transactions on Computers. Volume C-17, Number 8 (August 1968) pp. 758-770.

[4] Lehman, M. "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors," Proceedings of the IEEE. December 1966, pp. 1889-1901.

[5] Fulmer, L. C., and W. C. Meilander. "A Modular Plated Wire Associative Processor," Proceedings of the IEEE Computer Group Conference. June 1970, pp. 325-335.

[6] Graham, W. R. "The Parallel and the Pipline Computers," Datamation. Volume 16, Number 4 (April 1970), pp. 68-71.

[7] Bremer, J. W. "A Survey of Mainframe Semiconductor Memories," Computer Design. Volume 9, Number 5 (May 1970), pp. 63-73.

[8] Texas Instruments Incorporated. The Integrated Circuits Catalog for Design Engineers. First edition.

[9] Yasui, T. "Pattern Matching Problem-Benchmark on ILLIAC IV," ILLIAC IV Document Number 217. University of Illinois at Urbana-Champaign. May 1970.

[10] Lincoln, N. R. "Parallel Programming Techniques," presented at the "SIGPLAN Symposium on Compiler Optimization." University of Illinois at Urbana-Champaign. July 1970.

[11] Wilhelmson, R., et. al. "Matrix Operations on ILLIAC IV," ILLIAC IV Document Number 52. University of Illinois at Urbana-Champaign. March 1967.

[12] Stevens, J. E., "Matrix Multiplication Algorithm for ILLIAC IV," ILLIAC IV Document Number 231. University of Illinois at Urbana-Champaign. August 1970.

[13] Troyer, S. "Sparse Matrix Multiplication," ILLIAC IV Document Number 137. University of Illinois at Urbana-Champaign. June 1968.

[14] Carr, R. "Gauss-Seidel on ILLIAC IV," ILLIAC IV Document Number 67. University of Illinois at Urbana-Champaign. May 1967.

[15] Ackins, G. "Fast Fourier Transform," ILLIAC IV Document Number 146. University of Illinois at Urbana-Champaign. July 1968.

[16] Stevens, J. "Fast Fourier Transform Subroutine for ILLIAC IV," ILLIAC IV Document Number 226, University of Illinois at Urbana-Champaign. July 1970.

[17] McIntyre, D. "ILLIAC IV Language Evaluation - A Preliminary Experiment," ILLIAC IV Document Number 213. University of Illinois at Urbana-Champaign. November 1970.

VITA

Born in 1941 in Santos, BRAZIL, Nelson Castro Machado received in December 1964 the degree of "Electronic Engineer" from the Instituto Tecnológico de Aeronáutica in São José dos Campos, BRAZIL. He then worked there for one and one half years as a teaching assistant, being responsible for courses in applied electronics, pulse circuits laboratory and automata theory. In September 1966 he came to the University of Illinois where he received the M.S. degree in October of 1969. Since his arrival in the U.S.A., Mr. Machado has been working as a research assistant, initially with the ILLIAC IV Project and later with the Center for Advanced Computation of the University of Illinois. In this activity, he was responsible for the semantics part of a Translator Writing System developed to help implement ILLIAC IV languages. His M.S. thesis entitled "ISL-A semantics Language for a Translator Writing System is a result of this research. From January 1970 until January 1972, Mr. Machado worked on the topic of parallel computer organization, exploring new approaches to the concept of array processor utilized in ILLIAC IV. This activity resulted in his Ph.D. dissertation entitled "An Array Processor with a Large Number of Processing Elements."

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

ORIGINATING ACTIVITY (Corporate author)

Center for Advanced Computation
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

2a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED

2b. GROUP:

REPORT TITLE

AN ARRAY PROCESSOR WITH A LARGE NUMBER OF PROCESSING ELEMENTS

DESCRIPTIVE NOTES (Type of report and inclusive dates)

Research Report

AUTHOR(S) (First name, middle initial, last name)

Nelson C. Machado

REPORT DATE: January 1, 1972

7a. TOTAL NO. OF PAGES: 184

7b. NO. OF REFS: 17

CONTRACT OR GRANT NO.: DAHC04-72-C-0001

PROJECT NO.: ARPA Order 1899

9a. ORIGINATOR'S REPORT NUMBER(S): CAC Document No. 25

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report): UIUCDCS-R-72-499

DISTRIBUTION STATEMENT

Copies may be obtained from the address given in (1) above. Distribution unlimited; approved for public release.

SUPPLEMENTARY NOTES: None

12. SPONSORING MILITARY ACTIVITY: U.S. Army Research Office-Durham
Durham, North Carolina

ABSTRACT

This paper describes a new type of array processor (SPEAC) which could be characterized as an intermediate between ILLIAC IV and the Associative Processor. The number of processing elements (PE's) is typically 1K but could go as high as 8K. Each PE is a relatively simple unit with about 1K equivalent gates, designed to allow implementation either on a single very complex LSI chip or on several MSI chips. Each PE plus its memory (PEM) could then be assembled on one single printed circuit board or ceramic substrate.

Processing is performed in groups of four bits which allows variable word length. Maximum freedom in data format and instruction format is made possible by the use of a microprogrammable control unit (CU). Therefore, the machine is quite versatile and can be used efficiently either on floating-point large precision problems (matrix operations, signal processing, etc.) or on fixed-point small precision ones (character manipulation, picture processing, etc.).

PE design is carried out in great detail and a general sketch of the CU is presented. Operations are described and timed, with particular emphasis on floating-point addition (20 μsec per PE for 32 bits) and floating-point multiplication (25 μsec per PE for 32 bits). A few typical applications are presented along with their time estimates.

D FORM 1 NOV 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Design and Construction						
General Purpose Computer						
Arithmetic Units						

IOGRAPHIC DATA T	1. Report No. UIUCDCS-R-72-499	2.	3. Recipient's Accession No.
le and Subtitle	AN ARRAY PROCESSOR WITH A LARGE NUMBER OF PROCESSING ELEMENTS		5. Report Date January 1, 1972
			6.
hor(s)	Nelson Castro Machado		8. Performing Organization Rept. No.
forming Organization Name and Address	Center for Advanced Computation University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. Project/Task/Work Unit No.
			11. Contract/Grant No. DAHC04-72-C-0001
nsoring Organization Name and Address	U.S. Army Research Office-Durham Duke Station Durham, North Carolina		13. Type of Report & Period Covered Research
			14.

Supplementary Notes

None

Abstracts

See DD Form Number 1473.

Words and Document Analysis. 17a. Descriptors

Design and Construction

General Purpose Computer

Arithmetic Unit

ntifiers/Open-Ended Terms

ATI Field/Group

ability Statement	19. Security Class (This Report)	21. No. of Pages
ies may be obtained from the address in (9) ve. Distribution unlimited.	UNCLASSIFIED	184
	20. Security Class (This Page) UNCLASSIFIED	22. Price NC

5-35 (10-70)

USCOMM-DC 40329-P71